

Docket No.: 57454-308

PATENT

7-21-02
#3ed
JCE 012 U.S. PRO
09/995837



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of

Mamoru SAKAMOTO, et al.

Serial No.:

Group Art Unit:

Filed: November 29, 2001

Examiner:

For: DATA PROCESSOR HAVING TRANSLATOR AND INTERPRETER THAT
EXECUTE NON-NATIVE INSTRUCTIONS

**CLAIM OF PRIORITY AND
TRANSMITTAL OF CERTIFIED PRIORITY DOCUMENT**

Commissioner for Patents
Washington, DC 20231

Sir:


In accordance with the provisions of 35 U.S.C. 119, Applicant hereby claims the priority of:

Japanese Patent Application Number 2000-368729, Filed December 4, 2000

cited in the Declaration of the present application. A Certified copy is submitted herewith.

Respectfully submitted,

MCDERMOTT, WILL & EMERY


Stephen A. Becker
Registration No. 26,527

600 13th Street, N.W.
Washington, DC 20005-3096
(202) 756-8000 SAB:kjw
Date: November 29, 2001
Facsimile: (202) 756-8087

57454-308
Mamoru Sakamoto, et al
November 29, 2001
McDermott, Will & Emery

日 本 国 特 許
PATENT OFFICE
JAPANESE GOVERNMENT

J6872 U.S. PRO
09/995837
11/29/01

別紙添付の書類に記載されている事項は下記の出願書類に記載されて
いる事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed
with this Office.

出 願 年 月 日
Date of Application: 2000年12月 4日

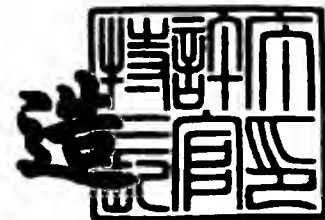
出 願 番 号
Application Number: 特願2000-368729

出 願 人
Applicant(s): 三菱電機株式会社

2000年12月22日

特許庁長官
Commissioner,
Patent Office

及川耕造



出証番号 出証特2000-3107568

【書類名】 特許願

【整理番号】 527365JP01

【提出日】 平成12年12月 4日

【あて先】 特許庁長官殿

【国際特許分類】 G06F 9/44
G06F 9/455

【発明者】

【住所又は居所】 東京都千代田区丸の内二丁目 2 番 3 号 三菱電機株式会
社内

【氏名】 坂本 守

【発明者】

【住所又は居所】 東京都千代田区丸の内二丁目 2 番 3 号 三菱電機株式会
社内

【氏名】 吉田 豊彦

【特許出願人】

【識別番号】 000006013

【氏名又は名称】 三菱電機株式会社

【代理人】

【識別番号】 100064746

【弁理士】

【氏名又は名称】 深見 久郎

【選任した代理人】

【識別番号】 100085132

【弁理士】

【氏名又は名称】 森田 俊雄

【選任した代理人】

【識別番号】 100091409

【弁理士】

【氏名又は名称】 伊藤 英彦

【選任した代理人】

【識別番号】 100096781

【弁理士】

【氏名又は名称】 堀井 豊

【選任した代理人】

【識別番号】 100096792

【弁理士】

【氏名又は名称】 森下 八郎

【手数料の表示】

【予納台帳番号】 008693

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【ブルーフの要否】 要

【書類名】 明細書

【発明の名称】 データ処理装置

【特許請求の範囲】

【請求項 1】 所定の命令群をネイティブコードとするプロセッサ、

前記プロセッサに対する非ネイティブコードを前記プロセッサの 1 または 2 以上のネイティブコードに変換するハードウェアトランスレータ、

前記プロセッサ上で動作し、前記プロセッサに対する非ネイティブコードを前記プロセッサの 1 または 2 以上のネイティブコードに変換するソフトウェアトランスレータ、

前記ソフトウェアトランスレータの出力するネイティブコードを記憶するための記憶手段、

前記プロセッサ上で動作し、前記プロセッサに対する非ネイティブコードを逐次解釈し、前記プロセッサのネイティブコードを用いて実行するソフトウェアインタープリタ、および

所定の基準にしたがって、前記ハードウェアトランスレータによって出力されるネイティブコードの実行、前記ソフトウェアトランスレータによって出力されるネイティブコードの実行、ならびに前記ソフトウェアインタープリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択して前記プロセッサを動作させるための選択手段とを含む、データ処理装置。

【請求項 2】 前記選択手段は、非ネイティブコードの種類または実行頻度、若しくは前記記憶手段の状態に依存して、前記ハードウェアトランスレータによって出力されるネイティブコードの実行、前記ソフトウェアトランスレータによって出力されるネイティブコードの実行、ならびに前記ソフトウェアインタープリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択して前記プロセッサを動作させるための手段を含む、請求項 1 に記載のデータ処理装置。

【請求項 3】 前記選択手段は、前記所定の基準にしたがって、前記ネイティブコードによって構成されるメソッドが呼出されるごとに、前記ハードウェアトランスレータによって出力されるネイティブコードの実行、前記ソフトウェア

トランスレータによって出力されるネイティブコードの実行、ならびに前記ソフトウェアインタプリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択して前記プロセッサを動作させるための手段を含む、請求項 1 に記載のデータ処理装置。

【請求項 4】 前記ソフトウェアトランスレータは、前記非ネイティブコードに含まれるメモリオペランドの少なくとも一部を、前記プロセッサに備えられたレジスタに割り付けるように非ネイティブコードをネイティブコードに変換するためのコード変換手段を含む、請求項 1 に記載のデータ処理装置。

【請求項 5】 前記非ネイティブコードは、所定のスタックマシンのネイティブコードであり、

前記コード変換手段は、前記メモリオペランドのうちのスタックオペランドのうち、スタックトップ側のスタックオペランドが、前記プロセッサに備えられたレジスタに保持されるように、前記データ処理装置に接続されるメモリと、前記プロセッサのレジスタとの間で実行時にデータの保存と復帰とを行なうようなネイティブコードを生成するための手段を含む、請求項 4 に記載のデータ処理装置。

【請求項 6】 前記コード変換手段は、

スタックへのデータの転送のみを行なう非ネイティブコードを検出してその転送元と転送先とを対応付けて記憶するための手段をさらに含み、

前記生成するための手段は、前記データの転送のみを行なう非ネイティブコードの検出前記メモリオペランドのうちのスタックオペランドのうち、スタックトップ側のスタックオペランドが、前記プロセッサに備えられたレジスタに保持されるように、前記データ処理装置に接続されるメモリと、前記プロセッサのレジスタとの間で実行時にデータの保存と復帰とを行なうように、かつ、前記転送先をオペランドとして使用するコードについては、前記転送先に代えて、前記記憶するための手段に記憶された前記転送元をオペランドとするように、ネイティブコードを生成するための手段を含む、請求項 5 に記載のデータ処理装置。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】

本発明は、データ処理装置に関わるもので、特に、データ処理装置に搭載されているプロセッサのネイティブコードに加えて、非ネイティブコードを実行するための機能を備えたデータ処理装置に関わるものである。

【0002】

【従来の技術】

プロセッサはあるアーキテクチャに基づいて設計され製作される。アーキテクチャでは命令体系、命令形式などが決定され、そうした命令形式の命令を効率よく実行するようにハードウェアが製作される。データ処理装置において実行させるプログラムは、このように搭載されているプロセッサの命令コード（「ネイティブコード」と呼ぶ。）で記述しておくのが通常である。

【0003】

ところがこれに反して、搭載されているプロセッサ以外のプロセッサの命令コード（「非ネイティブコード」と呼ぶ。）で記述されているプログラムを実行させたいという場合もある。ひとつの例は、新しいプロセッサ用のプログラム開発が間に合わない際に、古いプロセッサ用のプログラムを新しいプロセッサで実行させたいという場合である。別の例は、Java（「Java」はSun Microsystems, Incの商標である。）バイトコードのように、仮想的なプロセッサ（「Java仮想マシン」と呼ばれる。）用の言語で記述されたひとつのプログラムを、それぞれ異なるプロセッサを搭載した複数種類の装置で実行させたいという場合である。

【0004】

非ネイティブコードをプロセッサで実行するための手段として従来から用いられている手法には、ソフトウェアインタプリタ、ソフトウェアトランスレータ、ハードウェアトランスレータがある。

【0005】

ソフトウェアインタプリタを用いる手法では、以下の一連の処理ステップをプロセッサ上でソフトウェアインタプリタと呼ばれるソフトウェアを実行することにより行なう。

【0006】

- (1) 非ネイティブコードのメモリからの読み出し
- (2) 読み出した非ネイティブコードに対する処理ルーチンへのディスパッチ
- (3) 読み出した非ネイティブコードに対する処理ルーチンの実行
- (4) 非ネイティブコードのプログラムカウンタの更新

このソフトウェアインタプリタ自体は、搭載されているプロセッサのネイティブコードで記述されている。

【0007】

ソフトウェアは柔軟性が高く、処理速度を考慮しなければ實際上どのような処理でも実現できるため、このソフトウェアインタプリタは容易に実現可能である。しかしその反面、ステップ(3)での実際の処理以外に、(1)、(2)および(4)の追加処理が必要となるため、実行速度が低下するという問題がある。

【0008】

一方、「トランスレータ」とは、非ネイティブコードのプログラムを、同等内容のネイティブコードのプログラムに変換する装置のことをいう。変換をハードウェアで行なうものがハードウェアトランスレータであり、ソフトウェアで行なうものがソフトウェアトランスレータである。

【0009】

ハードウェアトランスレータについては、たとえば、米国特許第5875336号に述べられている。ハードウェアトランスレータは、非ネイティブコードの各命令の動作をシミュレーションするためにハードウェアにより、非ネイティブコードの各コードと同等内容の処理を実現するネイティブコードを生成する。しかしこの際、以下の要因によって変換後のネイティブコードの実行速度が低下するという問題がある。

【0010】

第1に、トランスレータでは、非ネイティブコードの読出のために、演算結果だけでなく非ネイティブコードのPC（プログラムカウンタ）値または必要に応

じてフラグまでシミュレーションしなければならない。そのため、1つの非ネイティブコードの動作を多数のネイティブコードで置き換えることになってしまう。

【0011】

第2に、プロセッサに備えられるレジスタ数として非ネイティブコードが想定している数よりも、プロセッサに備えられているレジスタの数のほうが多い場合であっても、それを利用することができない。たとえば、トランスレータでは、非ネイティブコードのメモリオペランドはネイティブコードにおいてもやはりメモリオペランドに変換され、レジスタには割当てられない。

【0012】

変換後のネイティブコードの実行速度が低下するという問題に対する解決策の一つが米国特許第5898885号において提案されている。この手法は、非ネイティブコードがJavaバイトコードのようなスタックマシンのコードであることを前提としている。提案された手法では、スタックヘデータをプッシュする非ネイティブコードにそのデータをポップする非ネイティブコードが連続するような場合に、それらをまとめて実行できるようなネイティブコードを生成する。これにより、メモリへのアクセス回数が減少するので、実行速度の高速化を図ることができる。米国特許第6026485号にも類似の技術の提案がある。

【0013】

しかし、この方法は、その性質上、プッシュの直後にそのデータのポップがなければ適用することができないという制限がある。

【0014】

ソフトウェアトランスレータにより変換する場合も、上記のような速度低下という問題はある。しかし、ハードウェアトランスレータとは異なりソフトウェアトランスレータでは命令単位ではなくより大きなプログラム単位（サブルーチン単位やクラス単位）での変換処理を行うことができるという柔軟性があるため、無駄なメモリアクセスなどをなくすることが可能で、速度低下をある程度に押さえることができる。しかし、実行時の速度低下が少ないネイティブコードを生成しようとするれば、変換処理が複雑となり、変換処理時間が増大するという問題があ

る。

【0015】

このような変換処理時間のオーバーヘッドを押さえるため、ソフトウェアトランスレータでは、一度変換して生成したネイティブコードをメモリに保持しておく手法が用いられる。再度同じプログラム部分が実行されるときには、変換処理を行なうことはせず、メモリに保持されている変換後のネイティブコードが再利用される。しかしこうした、非ネイティブコードのプログラムを変換した後のネイティブコードを保持するために大容量のメモリを新たに必要とし、メモリのコストが増大してしまうという別な問題が発生する。

【0016】

変換後のネイティブコードを保持するためのメモリ容量の増大を防ぐために、変換後のネイティブコード保持用のRAM (Random Access Memory) を一定サイズとしておき、ソフトウェアキャッシュとして用いる手法を考えることができる。この手法では、サブルーチン (またはメソッド) 単位に非ネイティブコードをネイティブコードに変換してこのRAMに追加保持していく。このRAMが一杯になったときには、実行が終わってそれ以後に実行する可能性が低いサブルーチン (あるいはメソッド) のネイティブコードをRAMから解放する。解放してできた空きRAM領域に新たな変換後のネイティブコードからなるサブルーチン (あるいはメソッド) を保持する。

【0017】

このようなソフトウェアキャッシュを併用するソフトウェアトランスレータは、メモリ容量増大の防止に対してはある程度効果が期待される。しかし、メモリに全ての変換後の命令を保持する場合と比較して、変換処理の回数が増えるため、そのオーバーヘッドの増大が問題となる。変換処理時間を短く押さえるために変換処理の内容を単純化し非ネイティブコード単位で変換すると、変換後生成されたネイティブコードの実行速度が低くなってしまうということになるのは既に述べたとおりである。

【0018】

【発明が解決しようとする課題】

以上のように、ソフトウェアインタープリタでは実行速度が極端に遅くなるという問題がある。一方、ハードウェアトランスレータでもソフトウェアインタープリタほどではないにしろ、実行速度の低下を防止することがある程度以上は困難である。また非ネイティブコード全てをトランスレートするためにはハードウェア量が増大してしまうという問題がある。また、ソフトウェアトランスレータには実行速度の向上とメモリ増加の防止とが両立しにくいという問題がある。

【0019】

本発明は上記のような従来の方法に残る課題を解決するためになされたもので、非ネイティブコードを少ないハードウェア量で高速に実行することが可能な、非ネイティブコードのトランスレータを有するデータ処理装置を提供することを目的とする。

【0020】

【課題を解決するための手段】

請求項1に記載の発明にかかるデータ処理装置は、所定の命令群をネイティブコードとするプロセッサ、プロセッサに対する非ネイティブコードをプロセッサの1または2以上のネイティブコードに変換するハードウェアトランスレータ、プロセッサ上で動作し、プロセッサに対する非ネイティブコードをプロセッサの1または2以上のネイティブコードに変換するソフトウェアトランスレータ、ソフトウェアトランスレータの出力するネイティブコードを記憶するための記憶手段、プロセッサ上で動作し、プロセッサに対する非ネイティブコードを逐次解釈し、プロセッサのネイティブコードを用いて実行するソフトウェアインタープリタ、ならびに所定の基準にしたがって、ハードウェアトランスレータによって出力されるネイティブコードの実行、ソフトウェアトランスレータによって出力されるネイティブコードの実行、ならびにソフトウェアインタープリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択してプロセッサを動作させるための選択手段とを含む。

【0021】

変換後のネイティブコードを記憶するための記憶手段を必要としないが、処理速度の点で劣るハードウェアトランスレータおよびソフトウェアインタープリタ

と、記憶手段を必要とするが処理速度の点で優るソフトウェアトランスレータとを、所定の基準にしたがって選択し使い分けることができる。そのため、非ネイティブコードを、比較的少ない容量の記憶手段を用いて実行できる。ソフトウェアトランスレータを用いると処理が高速に行なえるので、全体としての処理の高速化が可能となる。また、ハードウェアトランスレータのハードウェア量を増大させるような命令についてはソフトウェアインタプリタで実行できるので、ハードウェアトランスレータは、限定された数の非ネイティブコードを変換するだけでよい。そのハードウェアは比較的単純となる。またソフトウェアインタプリタで実行される命令は限定されるので、その処理が遅くとも全体の処理速度にはそれほどの影響は生じない。

【 0 0 2 2 】

請求項 2 に記載の発明にかかるデータ処理装置は、請求項 1 に記載の発明の構成に加えて、選択手段は、非ネイティブコードの種類または実行頻度、若しくは記憶手段の状態に依存して、ハードウェアトランスレータによって出力されるネイティブコードの実行、ソフトウェアトランスレータによって出力されるネイティブコードの実行、ならびにソフトウェアインタプリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択してプロセッサを動作させるための手段を含む。

【 0 0 2 3 】

非ネイティブコードの種類、実行頻度、記憶手段の状態に依存していずれかを選択するため、非ネイティブコードの実行時の結果に基づいて適切にどのトランスレータまたはインタプリタを選択するかを決定できる。そのため、ハードウェア量をより少なく、および／または全体の処理速度をより高くすることができる。

【 0 0 2 4 】

請求項 3 に記載の発明にかかるデータ処理装置は、請求項 1 に記載の発明の構成に加えて、選択手段は、所定の基準にしたがって、ネイティブコードによって構成されるメソッドが呼出されるごとに、ハードウェアトランスレータによって出力されるネイティブコードの実行、ソフトウェアトランスレータによって出力

されるネイティブコードの実行、ならびにソフトウェアインタープリタの実行による非ネイティブコードの逐次解釈および実行のいずれかを選択してプロセッサを動作させるための手段を含む。

【 0 0 2 5 】

メソッドの呼出し時ごとに、ハードウェアトランスレータ、ソフトウェアトランスレータおよびソフトウェアインタープリタのいずれかを動的に選択できる。データ処理装置の状況、プログラムの実行状況などにあわせてデータ処理装置自体がいずれを選択するかを決定できるため、ハードウェア量をより少なく、および／または全体の処理速度をより高くすることができる。

【 0 0 2 6 】

請求項 4 に記載の発明にかかるデータ処理装置は、請求項 1 に記載の発明の構成に加えて、ソフトウェアトランスレータは、非ネイティブコードに含まれるメモリオペランドの少なくとも一部を、プロセッサに備えられたレジスタに割り付けるように非ネイティブコードをネイティブコードに変換するためのコード変換手段を含む。

【 0 0 2 7 】

メモリアクセスがレジスタへのアクセスに置換されるため、時間を要するメモリアクセスが減少し、変換後のプログラムの実行を高速化できる。

【 0 0 2 8 】

請求項 5 に記載の発明にかかるデータ処理装置は、請求項 4 に記載の発明の構成に加えて、非ネイティブコードは、所定のスタックマシンのネイティブコードであり、コード変換手段は、メモリオペランドのうちのスタックオペランドのうち、スタックトップ側のスタックオペランドが、プロセッサに備えられたレジスタに保持されるように、データ処理装置に接続されるメモリと、プロセッサのレジスタとの間で実行時にデータの保存と復帰とを行なうようなネイティブコードを生成するための手段を含む。

【 0 0 2 9 】

スタックオペランドがメモリではなくレジスタに割当てられる。その結果、時間を要するメモリアクセスが減少し、変換後のプログラムの実行を高速化できる

【 0 0 3 0 】

請求項 6 に記載の発明にかかるデータ処理装置は、請求項 5 に記載の発明の構成に加えて、コード変換手段は、スタックへのデータの転送のみを行なう非ネイティブコードを検出してその転送元と転送先とを対応付けて記憶するための手段をさらに含み、生成するための手段は、データの転送のみを行なう非ネイティブコードの検出メモリオペランドのうちのスタックオペランドのうち、スタックトップ側のスタックオペランドが、プロセッサに備えられたレジスタに保持されるように、データ処理装置に接続されるメモリと、プロセッサのレジスタとの間で実行時にデータの保存と復帰とを行なうように、かつ、転送先をオペランドとして使用するコードについては、転送先に代えて、記憶するための手段に記憶された転送元をオペランドとするように、ネイティブコードを生成するための手段を含む。

【 0 0 3 1 】

変換後のネイティブコードの数が、このような処理を行なわない場合と比較して減少する。実行すべきコード数が減少するので、変換後のプログラムの実行を高速化できる。

【 0 0 3 2 】

【発明の実施の形態】

以下に述べる本発明の実施の形態にかかるデータ処理装置はいずれも、ソフトウェアにより J a v a バイトコードをネイティブコードに変換するソフトウェアトランスレータ、ハードウェアにより J a v a バイトコードをネイティブコードに変換するハードウェアトランスレータ、ソフトウェアで J a v a バイトコードを実行時に解析してシミュレートするソフトウェアインタプリタと、対象の J a v a バイトコードの内容に依存してこれら 3 つの手段を動的に切り替えるトランスレータを備えたものである。本発明がこれらの実施の形態に限定されるわけではないことは当業者には明らかである。たとえば、以下の各実施の形態の説明では、非ネイティブコードとして J a v a バイトコードを想定するが、それ以外のコードを非ネイティブコードとして実行する場合も同様の考えでトランスレー

タを実現することができる。

【0033】

第1の実施の形態

図1は、本発明の一実施の形態にかかるデータ処理装置の概略構成を示したものである。図1を参照して、このデータ処理装置は、互いにバス107に接続されたプロセッサ101、RAM102と、所定アドレスに非ネイティブコードを格納するROM (Read-Only Memory) 103とを含む。

【0034】

単一の半導体チップで構成されたプロセッサ101は、プロセッサ101の本体であり従来のプロセッサと同等の機能を持つ演算部106と、演算部106とバス107との間に挿入され、バス107から受取った命令がROM103の所定アドレスから読出された非ネイティブコードであればそれを演算部106のネイティブコードに変換して演算部106に与え、ROM103の所定アドレスから読出された非ネイティブコードでなければ、そのまま演算部106に与える処理を行なうための多機能命令デコーダ105とを含む。

【0035】

RAM102は、演算部106のネイティブコードのメソッドと、このメソッドが使用するデータとを格納するためのものである。

【0036】

ROM103は、非ネイティブコードで記述されたプログラムおよび後述する非ネイティブコードのメソッド呼出し処理ルーチンをプロセッサ101で実行するプログラム（プロセッサ101にソフトウェアトランスレータを動作させるためのプログラム、およびプロセッサ101にソフトウェアインタープリタを動作させるためのプログラムを含む）を記憶する、コンピュータ（プロセッサ101）に読取り可能な記憶媒体である。本実施の形態の装置では、非ネイティブコードとして、Java仮想マシンの命令セットであるJavaバイトコードを使用する。メソッド呼出し処理ルーチン自体はネイティブコードからなっている。

【0037】

演算部106は三菱電機株式会社製32ビットマイクロプロセッサであるM3

2 Rの演算部の機能を持つ。図2～図4にこの演算部106の命令セットを示す。M32Rプロセッサについては、「三菱32ビットシングルチップマイクロコンピュータM32Rファミリソフトウェアマニュアル (Rev. 1.1)」に詳細に説明されている。なお本実施の形態ではM32Rを使用しているが、これに限定されるものではなく、他のプロセッサの演算部であっても以下の説明から容易に同様のトランスレータを備えたデータ処理装置が実現可能であることは当業者には明らかである。

【0038】

図5を参照して、バス107は、データバス107Aとアドレスバス107Bとを含む。多機能命令デコーダ105は、データバス107Aに接続され、データバス107Aから与えられた非ネイティブコードを演算部106のネイティブコードに変換するためのハードウェアトランスレータ120と、ハードウェアトランスレータ120の出力を受ける第1の入力と、データバス107Aに接続された第2の入力とを有し、制御信号の値によっていずれか一方の入力の信号を選択して演算部106に与えるためのマルチプレクサ121と、アドレスバス107Bに接続され、アドレスバス107B上のアドレスを非ネイティブコードの格納領域を規定するアドレスと比較して、アドレスバス107B上のアドレスが非ネイティブコードの格納領域のアドレスであればハードウェアトランスレータ120の出力を選択するように、それ以外の場合にはデータバス107Aからの入力を選択するように、マルチプレクサ121を制御する制御信号を出力するための比較回路122とを含む。

【0039】

多機能命令デコーダ105は、ROM103中の、プログラムカウンタが指すアドレス（アドレスバス107B上）から読み出した命令が非ネイティブコードであればそれを演算部106のネイティブコードに変換して結果を演算部106に渡す。この処理は後述する図10を参照して説明する。

【0040】

図6は本実施の形態におけるJavaバイトコードの変換例を説明するための変換対象とするメソッドのJavaバイトコードを示したものである。Java

では、ソースプログラムから、J a v a の実行環境となる仮想マシン用の中間コードに変換を行ない、この中間コードを各マシンに用意された仮想マシン実行環境で実行する。この中間コードはバイト単位で可変長であるため、J a v a バイトコードと呼ばれる。なお、J a v a 仮想マシンおよび J a v a バイトコードについては、文献「T. Lindholm, F. Yellin, The J a v a (TM) 仮想マシン仕様」アジソン・ウェスレイ・パブリッシャーズ・ジャパン、星雲社に詳しく述べられている。

【0041】

図7は第1の実施の形態によるデータ処理装置のプログラム実行手順を示したフローチャートである。J a v a バイトコードで記述されたプログラムを実行するには、実行前準備処理301の後、最初に実行する J a v a バイトコードのメソッドを呼出し実行する(302)。なおここで、J a v a バイトコードではサブルーチンのことをメソッドと呼んでいるので、以下の説明でもサブルーチンのことは一般的にメソッドと呼ぶ。

【0042】

図8は図7のステップ302における J a v a バイトコードのメソッドの呼出し実行処理の手順を示したフローチャートである。以下の説明では、ソフトウェアトランスレータによる実行の場合は、J a v a バイトコードのメソッドをネイティブコードのサブルーチンに変換した場合、変換結果のネイティブコードを R A M 1 0 3 に設定されたソフトウェアキャッシュ内に格納しておくことを前提としている。この場合、同じ非ネイティブコードを呼出した場合、変換後のネイティブコードをソフトウェアキャッシュから読出して実行することにより、変換処理を軽減し実行速度を向上させている。

【0043】

したがって非ネイティブコードのサブルーチンの呼出し処理では、最初に以前にこのメソッドが呼び出されたときの変換結果が R A M 1 0 3 に残っているかどうか調べられ(401)、残っていれば変換した結果のネイティブコードのサブルーチンを R A M 1 0 3 から呼出して実行する(405)。

【0044】

このメソッドの変換結果がRAM103に残っていない場合は、ステップ402においてハードウェアトランスレータ、ソフトウェアトランスレータおよびソフトウェアインタプリタのどれを使用してこのメソッドを実行するかを選択する処理を行ない、選択された実行方法にステップ403で分岐する。

【0045】

ハードウェアトランスレータによる実行の場合は、非ネイティブコードをサブルーチンコールする(406)。あとは多機能命令デコーダ105がプログラムカウンタのアドレスから読み出した非ネイティブコードをネイティブコードに変換して演算部106がそれを実行する(図10)。

【0046】

ソフトウェアインタプリタによる実行が選択された場合には、読出されたコードを引数としてソフトウェアインタプリタサブルーチンをコールする処理を行なう(407)。このサブルーチンは非ネイティブコードを一コードずつ解釈して実行する。

【0047】

ソフトウェアトランスレータによる処理が選択された場合には、ステップ404で、呼び出そうとしている非ネイティブコードのサブルーチンをネイティブコードのサブルーチンに変換する処理を行なう。変換した結果はRAM103に一時ストアされ、この後、RAM103をアクセスして変換した結果のネイティブコードのサブルーチンを呼び出す処理が実行される(405)。

【0048】

図9は図8のステップ402におけるサブルーチン実行方法選択処理の処理手順を示したフローチャートである。ここでは、ソフトウェアキャッシュの状態と、実行しようとしているサブルーチンの実行頻度とからハードウェアトランスレータで実行するか、ソフトウェアトランスレータで実行するかを切り替えている。すなわち、実行しようとしているサブルーチンが呼び出された回数があらかじめ設定された一定の回数を超えている場合にはソフトウェアトランスレータで実行し、そうでない場合にはハードウェアトランスレータで実行する。この実施の形態では、この回数は、変換後のネイティブコードを格納しているソフトウェア

キャッシュに空きがある場合（この場合を「N」とする。）とない場合（この場合を「M」とする。）とで分けている。

【0049】

図9を参照して、このルーチンの最初ではソフトウェアキャッシュに空きがあるか否かを判定する（417）。空きがある場合には制御はステップ418に進み、空きがない場合には制御はステップ420に進む。

【0050】

ステップ418では、呼び出そうとしているルーチンの呼出し回数が前述したNより大きいかな否かを判定する。大きい場合には制御はステップ422に、それ以外の場合には制御はステップ419に進む。

【0051】

ステップ419では、ハードウェアトランスレータを使用することとし、図示しない、分岐制御のためのフラグをセットして処理を終了する。

【0052】

ソフトウェアキャッシュに空きがない場合には、ステップ420で呼び出そうとしているサブルーチンの呼出し回数が前述したMより大きいかな否かについて判定する。呼出し回数がMより大きい場合には制御はステップ422に、それ以外の場合には制御はステップ419に、それぞれ進む。

【0053】

ソフトウェアキャッシュに空きがあり、かつサブルーチンの呼出し回数がNより大きい場合、およびソフトウェアキャッシュに空きがなく、かつサブルーチンの呼出し回数がMより大きい場合には、ステップ422において、ソフトウェアキャッシュ中のネイティブコードのサブルーチンの一部を解放して、ソフトウェアキャッシュ中に空き領域を作成することを試みる。

【0054】

続くステップ423で、ステップ422の処理の結果空き領域が作成されたかな否かについての判定を行なう。空き領域が作成された場合にはソフトウェアトランスレータを使用することが決定され（421）、作成できなかった場合にはソフトウェアインタープリタを使用することが決定される（424）。いずれの場

合にも、処理の分岐のためのフラグに適切な値が設定されて処理を終了する。

【0055】

図10は多機能命令デコーダ105の処理を示したフローチャートである。図5および図10を参照して、アドレスバス107B上に出力されたプログラムカウンタのアドレス範囲があらかじめ設定された非ネイティブコードの格納された空間内であるかどうかを比較回路122により判定し(432)、そうであればプログラムカウンタの指すアドレスから読み出したコードを非ネイティブコードとしてハードウェアトランスレータ120によりネイティブコードに変換する処理を行なう(434)。結果のネイティブコードを演算部106に渡す(435)。図5の例では、データバス107A上のコードをハードウェアトランスレータ120によりネイティブコードに変換し、その結果をマルチプレクサ121で選択して演算部106に与える。

【0056】

ステップ432において、プログラムカウンタのアドレス範囲が非ネイティブコードの格納された空間内でないとは判定された場合には、プログラムカウンタから読出されたコードはネイティブコードであるから、そのまま演算部106に渡す(433)。図5の例では、マルチプレクサ121により、ハードウェアトランスレータ120の出力ではなくデータバス107B上のコードを選択して演算部106に渡す。

【0057】

図10のステップ434における非ネイティブコードのネイティブコードへの変換処理の概略は以下のとおりである。すなわち、通常の非ネイティブコードの場合には、一つの非ネイティブコードに対して、1または複数の、順次に実行されるネイティブコードを予め対応付けておく。さらに非ネイティブコードのオペランドと、ネイティブコード内のオペランドとの対応関係を予め定めておく。この際、非ネイティブコードがJavaのような仮想マシンである場合、スタックをメモリのスタック領域またはレジスタに割当てるようにする。そして、非ネイティブコードの1命令を実行することに相当する処理を行なうためのネイティブコードの数に応じて、プログラムカウンタの値を適切な数だけ加算する。これが

変換処理の基本である。

【0058】

図11と図12とには、例外として、複雑な処理を要する（すなわち変換後のネイティブコードのサイズが大きくなる）非ネイティブコードに対する変換処理と、メソッド呼び出しを行う非ネイティブコードinvokestaticに対する変換処理とを示している。invokestaticなど、非ネイティブコード（Javaバイトコード）については図64を参照。

【0059】

複雑な非ネイティブコード（faddなど）の場合には、図11に示すような処理で変換が行われる。すなわち、変換しようとする非ネイティブコードの処理を実行するためのソフトウェアルーチン（あらかじめROM103内に格納してある。）を呼び出すようなネイティブコードを生成する（436）。invokestaticに対しても図12に示されるように同様にその処理ルーチンを呼び出すが、ここで、この処理ルーチンはさらに図8で示された非ネイティブコードのメソッド呼出し実行処理を行うようになっている。したがって、一旦図12の処理が実行されると、その中で図8の処理が繰り返し実行されることで非ネイティブコードのプログラムが実行されていくことになる。このように非ネイティブコードの別のメソッドをコールする命令が一つのメソッドに存在する場合も含め、本実施の形態では、メソッドが呼び出される毎に図8に示す処理ルーチンが実行され、特に、ハードウェアトランスレータ、ソフトウェアトランスレータおよびソフトウェアインタプリタの選択が行なわれる。

【0060】

なお、ソフトウェアトランスレータを使用する場合も同様に、メソッド呼出しを行うJavaバイトコードに対応して変換後に生成されるネイティブコードは、図8で示された非ネイティブコードのメソッド呼出し実行処理を行うようになっている。図8が繰り返し実行されることで非ネイティブコードのプログラムが実行されていくことになる。

【0061】

ソフトウェアインタプリタを使用する場合も同様に、メソッド呼び出しを行

う J a v a バイトコードに対応して、図 8 に示された非ネイティブコードのメソッド呼出し実行処理を行い、図 8 の処理が繰り返し実行されることで非ネイティブコードのプログラムが実行されていくことになる。

【 0 0 6 2 】

なお、図 8 で示される非ネイティブコードのメソッド呼出し実行処理を行うルーチンは予め ROM 1 0 3 に格納されている。

【 0 0 6 3 】

図 1 3 は図 7 のステップ 3 0 1 における実行前処理の手順を示したフローチャートである。実行準備処理の内容は、プログラムが使用するスタック領域の R A M 1 0 2 への割当て (5 0 1) と、スタックポインタをレジスタに設定する処理 (5 0 2) とである。スタック領域の最終アドレスに 4 を加えた値をプロセッサ 1 0 における演算部 1 0 6 の S P レジスタに設定する。

【 0 0 6 4 】

図 1 4 は J a v a バイトコードのメソッドをネイティブコードのサブルーチンに変換する処理 (図 8 のステップ 4 0 4) の手順を示したフローチャートである。図 1 4 を参照して、まず変換開始処理を行い (6 0 1) 、レジスタ割り当てを行い (6 0 2) 、メソッド変換処理を行う (6 0 3) 。これら各処理について図 1 5 ～図 2 0 を参照して以下に述べる。

【 0 0 6 5 】

図 1 5 は変換開始処理 6 0 1 の手順を示したフローチャートである。図 1 5 を参照して、ステップ 7 0 1 で、変換対象の J a v a バイトコードの先頭アドレス、すなわちメソッドの実行開始アドレスを変数 jpcstart に代入し、最終アドレス + 1 を変数 jpcend に代入する。

【 0 0 6 6 】

続いて、ローカル変数とスタックオペランドとをメモリに割当てする (7 0 2) 。集合 R E G に、割当て可能なレジスタの番号を代入する (7 0 3) 。スタック深さリストを空にする (7 0 4) 。繰返し制御変数 i ($i = 0 \sim n S t a c k - 1$) に対して変数 $R S i$ を 0 にする (7 0 5) 。同様に繰返し制御変数 j ($j = 0 \sim n L o c a l - 1$) に対して変数 $R L j$ を 0 にする (7 0 6) 。ただしここ

で `nStack` はスタックオペランドの数であり、`nLocal` はローカル変数の数である。

【0067】

図16は図14に示すレジスタ割当て処理602の手順を示したフローチャートである。図16に示す処理では、アドレス `jpc` を `jpcstart` から `jpcend` まで変化させて、メソッド中の `Java` バイトコードについて順にステップ804～812を繰り返し処理する。

【0068】

まずステップ801で、スタック深さの登録リストを空にする。`jpc=jpcstart` の時点、すなわちメソッド実行開始時点でのスタック深さは0であるので、ステップ802では `Java` バイトコードのアドレス `jpcstart` とそのアドレスでのスタック深さとを互いに対応付けて `[0,0]` のような形でスタック深さリスト（図示せず）に登録している。この処理の内容は後述する。ステップ803で `jpc` を `jpcstart` の値に初期化する。

【0069】

ステップ804からステップ813までがアドレス `jpc` を `jpcstart` から `jpcend` まで変化させて行なう繰り返し処理である。

【0070】

ステップ804～806では `jpc` 実行時のスタックの深さ `js`（アドレス `jpc` の `Java` バイトコードが実行終了した時点でのスタックポインタが示すスタックの位置）を求めている。ステップ804で、アドレス `jpc` に対応付けて登録されたスタック深さがあるか否かを判定する。なければ、ステップ805で `js` に `jsnext` を代入してステップ807に進む。あれば、ステップ806で `jpc` に対応付けて登録されているスタック深さを `js` に代入し、ステップ807に進む。

【0071】

ステップ807で次のアドレス `jpcnext` を求める。具体的には、ステップ807でアドレス `jpc` にある `Java` バイトコードを `jinst` に代入し、`jinst` のコードサイズを変数 `jinstsize` に代入し、`jpc+jinstsize` を `jpcnext` に代入する。ステップ807で求められる `jinst` が `jpc` における `Java` バイトコード、`jinstsize` が

その J a v a バイトコードのサイズ (バイト単位) である。

【0072】

jpcでのスタック深さがjpcであるとき、jinst実行後のスタック深さは

「js - jinstがスタックからポップするオペランド数」 + 「jinstがスタックに
プッシュするオペランド数」

として求められる。そこで、ステップ808ではjinstがスタックからポップするオペランド数を変数consumeに、jinstがスタックにプッシュするオペランド数を変数produceに、それぞれ代入する。続いてステップ809で、js-consume+produceを変数jsnextに代入する。

【0073】

ステップ810ではこのようにして求めたjsnextをjinstの次に実行する可能性のあるアドレス (次の J a v a バイトコードのアドレスと、分岐先のアドレス) に対応して、スタック深さとして記録させておく。ステップ811ではjinstが参照するオペランドの参照カウントを記録している。ステップ812でjpcを求めた次のアドレスに更新する。さらにステップ813でjpc<jpcendか否かを判定し、jpcがjpcendになるまでステップ804からステップ812までの処理をループさせている。

【0074】

ステップ811で求められた参照カウント情報を元にステップ814でレジスタ割り当てを行っている。

【0075】

図17は、図16のステップ802、810における、アドレスとスタック深さの登録処理の手順を示したフローチャートである。最初に、登録しようとしているアドレスを変数jpcrecordに代入し、登録しようとしているスタック深さを変数jsrecordに代入する。続いてアドレスjpcrecordに対応したスタック深さが既に登録されているかどうかをチェックし (902)、されていないならアドレスjpcrecordとスタック深さjsrecordとの対応を登録する (905)。登録済み

なら登録済みの値と今登録しようとしている値jsrecordとを比較する（903）。両者が一致するなら処理を終了する。両者が一致しなければ、実行する経路によってスタック深さが異なるということであるが、このようなケースではスタックオペランドをレジスタに対応付けて割当てすることはできない（904）。したがってこの場合には図示しない、対応付け不可のフラグを設定しておく。

【0076】

図18は図16のステップ811におけるオペランド参照カウンタの記録処理の手順を示したフローチャートである。最初に変数jsminにjs - consumeの値を代入する（1001）。ポップするスタックオペランドについて参照カウンタRSiを+1し（1002）、プッシュするスタックオペランドについて参照カウンタRSiを+1している（1003）。また、ローカル変数を参照するか否かを判定し（1004）、参照するならば（1004の判定結果が「YES」）、そのローカル変数の参照カウンタRLiを+1する（1005）。

【0077】

図19は図16のステップ814におけるレジスタ割当ての決定処理の手順を示したフローチャートである。最初に、図17のステップ904でスタックオペランドのレジスタ割当てができないメソッドと判定されたか否かをフラグを参照することにより判定する（1101）。割当てが可能なメソッドであれば、参照カウンタRSiとRLiをソートして（1104）、大きい順にレジスタを割当てる（1005）。スタックオペランドをレジスタに対応付けて割当てることができないメソッドであることが判明している場合には（1101の判定結果が「YES」）、ローカル変数についてのみ同様にしてレジスタに割当てる（1102、1103）。

【0078】

図20は図14のステップ603におけるメソッド変換処理の手順を示したフローチャートである。まずメソッドのエントリにおいてネイティブコードを生成する。これに付いては後述する図37の（1）に示している（1201）。次に変数jpcの初期化処理としてjpcstartの値を代入する（1202）。

【0079】

以下、メソッド中の J a v a バイトコードについて順にステップ 1 2 0 3 ~ 1 2 0 7 を繰り返し行う。すなわち、アドレス jpc にある J a v a バイトコードを jinst に代入する (1 2 0 3)。次に変数 jinstsize に jinst のコードサイズを代入する (1 2 0 4)。そして jpc と jinstsize とを加算した結果を jpcnext に代入することで次のコードのアドレスを求める (1 2 0 5)。さらに、アドレス jpc に対応して登録されている、図 1 6 に示した処理で求められたスタック深さを変数 js に代入する (1 2 0 6)。そして、このようにして求められた jpc、jinst および js をもとにネイティブコードを生成している (1 2 0 7)。続いてステップ 1 2 0 8 で jpcnext を変数 jpc に代入し、さらにステップ 1 2 0 9 で jpc の値が jpcend より小さければ制御をステップ 1 2 0 3 に戻して以上の処理を繰り返し、jpc の値が jpcend より小さければ処理を終了する。

【0080】

ステップ 1 2 0 7 では J a v a バイトコードの種類に応じて図 2 1 ~ 図 2 8 のようなネイティブコードが生成される。これら図および以下の全図において、「TX」は X 番地の J a v a バイトコードに対して生成されたネイティブコードのアドレスを指す。各 J a v a バイトコードが対象とするオペランドがレジスタに割当てられているかメモリに割当てられているかに応じて生成するバイトコードを変化させている。

【0081】

なお、図 2 7 における「エピローグのコード」については、以降の実際の変換例の説明において詳細が示される。図 2 8 における引数をプッシュするコードについても、以降の実際の変換例の説明において詳細が示される。図 2 8 において、methodId は、呼び出そうとするメソッドを表す文字列が格納されている領域のアドレスである。callJavaMethod は、レジスタ R0 が指す領域に格納された文字列で指定されるメソッドを呼び出すための処理を図 8 の手順に従って行うようなルーチンである。

【0082】

図 2 9 はネイティブコードのサブルーチンにおける M32R のレジスタの使用方法を示したものである。本実施の形態ではサブルーチンの引数はスタックに格

納して渡すようにしているが、一部の引数をたとえばr0-r3レジスタに格納して渡すようにすることも可能である。

【0083】

図30～図33はネイティブコードのサブルーチンにおけるスタックの使用法を示したものである。図30はサブルーチンが呼び出された時点でのスタックの状態である。引数がスタック上に格納されている。図31は、サブルーチンの最初のエントリコードを実行した後の状態である。図30の状態から、さらに、ローカル変数用の領域とスタックオペランド用の領域が確保され、サブルーチン実行前後で保存しなければならないレジスタ(r8-r14)の値がプッシュされる。図中、nStackはスタックオペランドの数、nLocalはローカル変数の数、nArgは引数の数である。なお、nLocalには引数の数も含まれている。これらを図6の例に当てはめるとnStack=6、nLocal=3、nArg=2であり、スタック使用状況は図32、図33のようになる。

【0084】

以下、図6で示したJavaバイトコードのメソッドを図14の処理手順に従ってネイティブコードのサブルーチンに変換する処理について説明する。

【0085】

ステップ601の実行後、すべてのオペランドは図33で示すメモリ領域に割当てられる。この割当ての状態を図34に示す。たとえばローカル変数[0]（第1引数）は、スタックポインタからのオフセットが60バイトにある4バイトのメモリ領域に割当てられている。

【0086】

次にステップ602では、図16で示す処理が実行される。図35および図36はこの処理を実行中の様子を示したものである。

【0087】

ステップ801の開始直前にはオペランドの参照カウントRS0～RS5、RL0～RL2はすべて0に設定されている。（RS0～RS5はスタックオペランド[0]～[5]の参照カウントであり、RL0～RL2はローカル変数[0]～[2]の参照カウントである。）ステップ801、802により、[アドレス0、ス

タック深さ0]がスタック深さリストに登録される。(スタック深さリストは図示していないが、たとえば登録された順番をインデックスとしアドレス値とする配列と、登録された順番をインデックスとしスタック深さを値とするような配列とで容易に実現することができる。)次にステップ803により、jpc=0、jsnext=0に設定される。以上が状態(1)に示した段階である。

【0088】

状態(2)は、その後ステップ804~813を実行してステップ813のyesの側に達した時点での状態を示している。まず、jpcは以前の状態(1)での値0を引き継いでいる。次に、スタック深さリストに[アドレス0、スタック深さ0]が登録されているため、ステップ804、806を実行してjs=0となる。ステップ807で求められるjinstはiload_0、jinstsizeは1、jpcnextは1となる。iload_0命令はローカル変数[0]の値をスタックにプッシュするものであるため、ステップ808で求められるconsume, produce はそれぞれ0、1である。したがって、ステップ809で求められるjsnextはjs-consume+produce=0-0+1=1となる。iload_0の次に実行する可能性のあるのは次のアドレスjpcnext(=1)のJavaバイトコードだけであるため、ステップ810では、[アドレス1、スタック深さ1]が登録される。iload_0が参照するのはローカル変数[0](読み出し)とスタックオペランド[0](書き込み)であるため、ステップ811ではこれらに対応する参照カウンタRL0とRS0がそれぞれ+1されて、それぞれ1、1になる。

【0089】

同様に、その後ステップ804~813を実行して再び813のyesの側に達する処理を繰り返すことで、状態(3)(4)...)のように進行していく。

【0090】

状態(8)では、ifge 21命令が次に実行する可能性のあるのは、次のアドレス(jpcnext)である9番地か、分岐先のアドレスである21番地のいずれかであるため、[アドレス9,スタック深さ1]と[アドレス21、スタック深さ1]のふたつがスタック深さリストに登録されている。

【0091】

状態（18）では、goto 28命令が次に実行する可能性のあるのは、分岐先のアドレスである28番地であるため、[アドレス28,スタック深さ2]が登録されている。

【0092】

状態（23）では、invokestatic命令が次に実行する可能性のあるのは、次のアドレスである28番地であるため、[アドレス28,スタック深さ2]を登録しようとするが、アドレス28に対するスタック深さは状態（18）において既に登録済みであるため、登録済みのスタック深さと登録しようとするスタック深さとの一致比較が行われる（ステップ902）。この場合は一致するため、ステップ904はスキップされてアドレスとスタック深さの登録処理は終了する。

【0093】

状態（25）では、ireturn命令が次に実行する可能性のあるアドレスは実行時にだけ決定しこの時点では不明であるため、スタック深さリストには何も登録されない。

【0094】

以上のようにして、最後の状態（25）のRS0～RS5、RL0～RL2で示されるようなオペランドの参照カウントカウントが求められる。また、状態（1）～（25）の「登録される命令アドレスとスタック深さ」欄で示されるようなスタック深さがスタック深さリストに登録されることになる。

【0095】

ステップ814では、図19に示される処理が行われる。ステップ1101ではステップ1104に進む。ステップ1104ではここまですべて求められているオペランド参照カウントRS0～RS5、RL0～RL2を降順にソートする。この例ではRS1、RS0、RS2、RS3、RS4、RL0、RL2、RS5、RL1の順になる。ステップ1105においてこの順にはじめの6つのオペランドにレジスタR8～R13を割当てて、図34に示すようなレジスタ割当て結果となる。すなわち、スタックオペランド[1]にレジスタR8が、スタックオペランド[0]にレジスタR9が、スタックオペランド[2]にレジスタR10が、スタックオペランド[3]にレジスタR11が、スタックオペランド[4]にレジスタR

1 2 が、ローカル変数[0]にレジスタR1 3が、それぞれ割当てられる。

【0 0 9 6】

以上でステップ6 0 2を終了し、次にステップ6 0 3の処理に移る。ステップ6 0 3では図2 0に示す処理が行われる。図3 7および図3 8はこの処理を実行するにつれて生成されるネイティブコードを示したものである。なおこの図において、S<0>～S<5>、L<0>～L<2>の記号は、図3 4に示したオペランドを示している。たとえば「ldi S<0>, #1」とは、「ldi R9, #1」の意味であり、「ld S<3>, @L<1>」とは「ld R10, @(56,sp)」の意味である。

【0 0 9 7】

まず、ステップ1 2 0 1では、状態(1)で示すようなネイティブコードが生成される。このネイティブコードの内容は、これを実行することによって、図3 2から図3 3へスタックレイアウトを変更するためのコードと、ローカル変数のうち、引数であってレジスタに割当てられているもの(この例ではL<0>)についてメモリからレジスタに値を読み込んでおくためのコードである。なお、nLocal、nStack、nArgについては、図3 0～図3 3で使用したのと同じ意味である。

【0 0 9 8】

ステップ1 2 0 2においてjpc=0となる。ステップ1 2 0 3において求められるアドレス0にあるJ a v aバイトコードjinstはiload__0である。ステップ1 2 0 4において求められるjinstsizeは1である。ステップ1 2 0 5において求められるjpcnextは1である。図3 5の(1)に示すようにアドレス0に対応付けてスタック深さ0が登録されているため、ステップ1 2 0 6において求められるjsは0となる。ステップ1 2 0 7では、変換しようとするJ a v aバイトコードjinstに応じて、図2 1～図2 8に示されるようなネイティブコードの生成が行われる。今の例の場合は、jinstはiload__0であるので図2 2に示すようなネイティブコードが生成される。スタック深さjs=0におけるオペランドはS<0>とL<0>であるが、図3 4に示すように、オペランドS<0>はレジスタR9に、オペランドL<0>はレジスタR1 3に割当てられているので、図2 2の最初の行で示されるm v S<0>, L<0>(すなわちm v R9, R1 3)というネイティブコードが生成される。以上が図3 7の(2)で示すネイティブコードが生成されるまでの処理である。

【0099】

以下同様にして、図37～図38の(3)～(25)で示すネイティブコードが生成される。ここで、(25)のireturn命令に対して生成されるエピローグのネイティブコードは、スタックレイアウトを図33から図32へ戻すような処理を行っている。

【0100】

以上のように、本発明の第1の実施の形態による非ネイティブコードのプログラム実行方式によれば、ローカル変数とオペランドスタックとがレジスタに割当てられた結果、従来これらがメモリに割当てられていた場合に比べて、変換後のネイティブコード数が減少するとともに、メモリアクセス頻度も減少し、その結果実行速度も向上する。

【0101】

第2の実施の形態

本発明の第2の実施の形態におけるデータ処理装置の構成は多くの部分が第1の実施の形態と共通である。

【0102】

まず図1～図18までは第1および第2の実施の形態は同じであるが、図16のステップ814におけるレジスタ決定処理は、この第2の実施の形態の装置では、図19ではなく、図39のフローチャートで示す手順に従って行われる。

【0103】

スタックオペランドに対してはNS個のレジスタを割当てる。今の例ではR8-R11の4つのレジスタを割当てるものであり、NS=4である。たとえばレジスタR8はスタックオペランド[4n] (n=0, 1, ...) に割当てられることになるが、実行中はスタック深さをjsとして、スタックオペランド[b]～スタックオペランド[js-1]がレジスタに保持されるようなネイティブコードが生成される。すなわちレジスタR8は、 $4n < js$ であるような最大のnに対するスタックオペランド[4n]を実行中は保持することになる。レジスタR8に割当てられているが保持されていないスタックオペランドは、メモリ中にいったん保存される。ローカ

ル変数に付いては第 1 の実施の形態と同様に割当てられる。すなわちローカル変数の参照カウントをソートし (2 7 0 1) 大きい順にレジスタに割当てる (2 7 0 4)。

【0 1 0 4】

また図 2 0 に示す処理の流れ自体は第 1 の実施の形態の場合と同じであるが、ステップ 1 2 0 7 における jinst の J a v a バイトコードに対するネイティブコードの生成処理は、図 2 1 ～図 2 8 ではなく、jinst の種類に応じて図 4 0 ～図 4 2 のフローチャートで示す手順に従って行われる。

【0 1 0 5】

また M 3 2 R レジスタの使用方法は図 2 9 ではなく図 4 3 で示すようになる。

図 4 0 は図 2 0 のステップ 1 2 0 7 において分岐しない J a v a バイトコードに対して行われるネイティブコード生成処理である。ステップ 2 8 0 1 では、使用されるスタックオペランドを求めており、[low] ～ [high-1] が使用されることになる。ステップ 2 8 0 2 ではこれらのオペランドのうち S<js> ～ S<high-1> について、割当てられたレジスタをこれらオペランド用にリザーブする処理を行う。レジスタに保持されているのはオペランド [b] ～ [js-1] であるので、これ以外のレジスタを使用する場合、[b-1] 以下のオペランドに付いてはメモリからレジスタにロードする処理 (2 8 0 3、2 8 0 4) を、[js] 以上のオペランドに付いてはレジスタをリザーブする処理 (2 8 0 2) を行っている。

【0 1 0 6】

ステップ 2 8 0 5 では第 1 の実施の形態の図 2 1 ～図 2 8 (スタックオペランドがレジスタの場合) で示したネイティブコードが生成される。ステップ 2 8 0 6、2 8 0 7、2 8 0 9 の処理は、実行経路の合流地点 (すなわち分岐先となるラベルのあるアドレス) において、スタックオペランドのスタックトップ側の N S (4) 個のエントリに割当てられたレジスタに値が保持されているようにするための処理である。スタックオペランドに割当てられたレジスタには必ずしも値を保持しているとは限らない (スタックトップ ～ スタックオペランド [b] がレジスタに保持されているが、b の値は合流点に到達する実行経路ごとに異なる可能性がある)。したがってそのままでは、複数の経路の実行パスが合流する地点に

において、レジスタの状態が不明となりそれ以降のネイティブコードの生成においてレジスタを利用することができなくなってしまう。そこで、実行の合流地点においては、必ずスタックオペランドのスタックトップ側の NS 個のスタックオペランドを割り当てられたレジスタに値を保持するようにしている。

【0107】

図41は、図20のステップ1207においてifgeに対して行われるネイティブコード生成処理である。ifgeについては図63の(8)を参照されたい。ifgeは $S<js-1>$ を参照するので、まずこれがレジスタに保持されていないなら(2901)、レジスタにロードする(2902)。最終的には条件分岐するコードを生成する(2906)が、その前に、2904、2905での合流用ネイティブコードを挿入する必要がある。この合流用ネイティブコードにおいて、レジスタ $S<js-1>$ にスタックオペランド $[js-5]$ の値を保持するようになる可能性があるので、前もって $S<js-1>$ をレジスタ $r0$ に移しておく(2903)。ステップ2904でアドレスjpcnextに対応付けて登録されたスタック深さを変数jpnnextに代入する。ステップ2905では、 $S<jsnext-1>$ から下の NS 個のスタックオペランドをレジスタに保存するためのネイティブコードを生成する。ステップ2906ではレジスタ $r0$ を元に条件分岐する命令を生成する。

【0108】

図42はステップ1207においてコードgotoに対して行われるネイティブコード生成処理である。コードgotoについては、図63の(18)を参照されたい。まず合流用ネイティブコードを生成する(3001、3002)。すなわち、この例では、分岐先のスタック深さを表すjsをjsnextに代入し(3001)、 $S<jsnext-1>$ から下の NS 個のスタックオペランドをレジスタに保持するネイティブコードを生成する(3002)。

【0109】

次に分岐命令(この例では「bra TX」)を生成する(3003)。次のアドレスにおいてはスタックトップの NS 個がレジスタに保持された状態を示すため、bを設定しなおす(3004、3005)。すなわち、jpcnextに対応付けて登録されたスタック深さを変数jsnextに代入する(3004)。次に、0とjsne

xt-NSとのうち大きい方の値を b に代入する (3 0 0 5)。

【0 1 1 0】

図 4 4 は図 4 0 のステップ 2 8 0 2 の処理を示している。この処理では $S\langle p \rangle \sim S\langle q \rangle$ レジスタのオーバーフロー対処のためのネイティブコードを生成する。そのため、 $i=p \sim q$ についてステップ 3 1 0 2 \sim 3 1 0 5 を繰り返している。

【0 1 1 1】

最初に、繰返し制御変数 i に p の値を代入する (3 1 0 1)。

繰返し部分では、 $S\langle i \rangle$ レジスタがスタックオペランド $[i-NS]$ の値を保持しているかどうかを調べる。 $[b] \sim [js-1]$ が対応するレジスタに保持されているわけなので、 $i-NS$ が $b \sim js-1$ の範囲にあれば保持されていることになる。この判定をステップ 3 1 0 2 および 3 1 0 3 で行なう。ステップ 3 1 0 2 において i が q 以下か否かを判定する。 i が q より大きければ処理を終了する。 i が q 以下であれば、ステップ 3 1 0 3 で、 $i-NS$ が b より小さいか否かを判定する。 $i-NS$ が b より小さければ制御はステップ 3 1 0 4 に進む、それ以外の場合にはステップ 3 1 0 4、3 1 0 5 の処理を行わずステップ 3 1 0 6 に進む。

【0 1 1 2】

$i-NS$ がレジスタに保持されていたら、メモリに保存するために「st $S\langle i-NS \rangle$, @SAVE $\langle i-NS \rangle$ 」というネイティブコードを生成する (3 1 0 4)。次に $i-NS$ がもうレジスタに保持されていないことを示すため、 b を $i-NS+1$ に変更する (3 1 0 5)。

【0 1 1 3】

図 4 5 は図 4 0 に示すステップ 2 8 0 4 の処理を示している。 $i=bottom \sim top$ について 3 2 0 4 を繰り返す。ステップ 3 2 0 1、3 2 0 3、3 2 0 5 の処理がその繰返し制御のための処理である。

【0 1 1 4】

すなわち、 $bottom$ が top 以下か否かを判定し、以下であればステップ 3 2 0 2 に進み、 $bottom$ が top より大きければ処理を終了する。

【0 1 1 5】

ステップ 3 2 0 2 で、 i に $bottom$ の値を代入する。ステップ 3 2 0 3 では、 i

がtop以下か否かを判定する。iがtop以下であれば制御はステップ3204に進む。それ以外の場合にはステップ3206でbottomまでレジスタに保持されていることを示すため、bにbottomの値を代入して処理を終了する。

【0116】

ステップ3204では、S<i>を復帰するために「ld S<i>, @SAVE<i>」というネイティブコードを生成する。ステップ3205でiに1を加算して、制御をステップ3203に戻す。以上の処理により、S<bottom>～S<top>レジスタのアンダフロー対処のためのネイティブコードが生成された。

【0117】

以下、図6で示したJavaバイトコードのメソッドを、第2の実施の形態における図14の処理手順に従ってネイティブコードのサブルーチンに変換する例について説明する。

【0118】

レジスタ決定処理814までは第1の実施の形態と同様に進行する。すなわち、ステップ601の実行後、すべてのオペランドは図46の変換開始処理直後の欄で示すようにメモリに割当てられる。これは第1の実施の形態の図34と同様である。また、ステップ602では、第1の実施の形態と同様に、図16で示した処理に従って図35で示すように処理が進行し、最後の状態(25)のRS0～RS5、RL0～RL2で示されるようなオペランドの参照カウントカウントが求められ、状態(1)～(25)の「登録される命令アドレスとスタック深さ」欄で示されるようなスタック深さが登録されることになる。

【0119】

ステップ814において、図39に示される処理が行われる。ステップ2701ではステップ2702に進む。ステップ2702では、スタックオペランドを4つのレジスタR8～R11に割り付ける。この例では、スタックオペランド[0]とスタックオペランド[4]とがレジスタR8に、スタックオペランド[1]とスタックオペランド[5]とがレジスタR9に、スタックオペランド[2]がレジスタR10に、スタックオペランド[3]がレジスタR11に、それぞれ割り付けられる。ステップ2703ではローカル変数の参照カウントRL0～RL2が降順にソー

トされる。この例ではRL0、RL2、RL1の順になる。ステップ2704においてこの順にはじめの2つのローカル変数にレジスタR12、R13を割当ててことで、図46に示すようなレジスタ割当て結果となる。

【0120】

以上でステップ602を終了し、次にステップ603の処理に移る。ステップ603では図20に示す処理が行われる。図47および図48はこの処理を実行するにつれて生成されるネイティブコードを示したものである。なおこの図において、S<0>～S<5>、L<0>～L<2>の記号は、図46に示すオペランドを示している。たとえばldi S<0>, #1とは、ldi R8, #1の意味であり、ld S<3>, @L<1>とはld R10, @(56, sp)の意味である。

【0121】

まず、ステップ1201では、状態(1)で示すようなネイティブコードが生成される。このコードの内容は、これを実行することによって、図32から図33へスタックレイアウトを変更するためのコードと、ローカル変数のうち、引数であってレジスタに割当てられているもの(この例ではL<0>とL<2>)についてメモリからレジスタに値を読み込んでおくためのコードである。

【0122】

次にステップ1202においてjpc=0となる。ステップ1203において求められるアドレス0にあるJavaバイトコードjinstはiload_0である。ステップ1204において求められるjinstsizeは1である。ステップ1205において求められるjpcnextは1である。図35の(1)に示すようにアドレス0に対応付けてスタック深さ0が登録されているため、ステップ1206において求められるjsは0となる。ステップ1207では、変換しようとするJavaバイトコードjinstに応じて、図40～図42に示すような処理が行われる。今の例の場合は、jinstはiload_0であるので、図40に示す処理が行われる。

【0123】

図40のステップ2801で求められるlow、highはそれぞれ0、1である。ステップ2802では図44に示す処理が行われるが、この場合はjs=0、b=0、low=0、high=1であるため、ステップ3101、3102(Yes)、3103

、3104 (No)、3107、3102 (No) と進行し、ネイティブコードを生成することなく終了する。次にステップ2803の判定が行われ、No側に進行する。ステップ2805の処理は、第1の実施の形態と同様の処理であり、図21～図28に示されている。今の例の場合は、図22に示すようなネイティブコードが生成される。ここではスタック深さ $js=0$ におけるオペランドは $S<0>$ と $L<0>$ であるが、図46に示すように、オペランド $S<0>$ はレジスタ $R8$ に、オペランド $L<0>$ はレジスタ $R12$ に割当てられているので、図22の最初の行で示される「mv $S<0>$, $L<0>$ 」(すなわち $mv\ R8, R12$) というネイティブコードが生成される。ステップ2806では $jpcnext (=1)$ にラベルがあるかどうか調べられるが、今の場合はラベルはないのでNoへ進んで図40の処理は終了する。以上が図47の(2)で示すネイティブコードが生成されるまでの処理である。

【0124】

以下同様にして、図47の(3)～(7)で示すネイティブコードが生成される。

【0125】

図47の(8)の変換処理では、ステップ1207において図41に示す処理が行われる。 $js=2$ 、 $b=0$ なのでステップ2901でNoへ進み、ステップ2903で「mv $r0, S<1>$ 」というネイティブコードが生成される。図35の(8)で示すようにアドレス9に対してはスタック深さ1が登録されているので、ステップ2904で求められる $jsnext$ は1である。ステップ2905では図49で示す処理が行われる。さらにステップ3301では図45に示す処理が行われる($bottom=MAX(0, jsnext-NS)=0$ 、 $top=b-1=-1$) が、ステップ3201における判定結果はNOとなり、処理を終了する。ステップ2906で「bge $r0, T21$ 」というネイティブコードが生成される。以上が(8)で示すネイティブコードが生成されるまでの処理である。

【0126】

図47の(9)～(11)に関しては(2)と同様の処理により、図47に示すネイティブコードが生成される。

【0127】

図 4 7 の (1 2) の変換処理では、ステップ 1 2 0 7 では図 4 0 に示す処理が行われる。ステップ 2 8 0 1 で求められる low、high はそれぞれ 4、5 である。ステップ 2 8 0 2 では図 4 4 に示す処理が行われるが、この場合は $p=4$ 、 $q=4$ 、 $b=0$ 、 $low=4$ 、 $high=5$ であるため、ステップ 3 1 0 3 における $i-NS=0$ となってステップ 3 1 0 4 が実行される。すなわちスタックオペランド [0] をメモリに保存するためのネイティブコードが生成される。今の例では、「st S<0>, @SAVE<0>」というネイティブコードが生成される。ここで、SAVE<0>は、レジスタ割当て処理を行う前に S<0> に対して割当てられていたメモリ領域を表している。今の例では、(4 0, sp) である。ネイティブコード生成後、ステップ 3 1 0 6 において b は 1 に更新される。ステップ 3 1 0 7 から、3 1 0 2 での判定が NO となり図 4 4 の処理は終了する。

【 0 1 2 8 】

図 4 0 に戻り、ステップ 2 8 0 3 からは No の側に進む。ステップ 2 8 0 5 の処理は、第 1 の実施の形態と同様の処理であるが、今の場合は J a v a バイトコードが `iconst_3` であるので、図 2 1 に示したネイティブコードが生成される。ここではスタック深さ $js=4$ におけるオペランドは S<4> であるが、オペランド S<4> はレジスタ R8 に割当てられているので、図 2 1 のオペランド割当てがレジスタの場合のネイティブコード、すなわち「ldi S<3>, #3」というネイティブコードが生成される。以上が図 4 7 の (1 2) で示すネイティブコードが生成される処理である。

【 0 1 2 9 】

図 4 7 の (1 3) の変換処理においては、(1 2) と同様にして図で示されるネイティブコードが生成される。ここではステップ 3 1 0 6 において b は 2 に更新される。

【 0 1 3 0 】

図 4 7 の (1 4) ～ (1 6) では、(2) と同様にして図で示されるネイティブコードが生成される。

【 0 1 3 1 】

図 4 7 の (1 7) の変換処理においては、ステップ 1 2 0 7 では図 4 0 に示す

処理が行われる。ステップ 2 8 0 1 で求められる low、high はそれぞれ 1、3 である。ステップ 2 8 0 2 では図 4 4 に示す処理が行われるが、この場合は $p=3$ 、 $q=2$ 、 $b=2$ 、 $low=1$ 、 $high=3$ であるため、ステップ 3 1 0 2 における判定結果が NO となり図 4 4 の処理は終了する。ステップ 2 8 0 3 ではステップ 2 8 0 4 に進む。ステップ 2 8 0 4 では図 4 5 に示す処理が行われる ($top=b-1=1$ 、 $bottom=low=1$)。ステップ 3 2 0 1、3 2 0 2、3 2 0 3 から $i=1$ となってステップ 3 2 0 4 が実行される。すなわちスタックオペランド [1] をメモリからレジスタに復帰するためのネイティブコードが生成される。今の例では、「ld S<1>, @SAVE<1>」といネイティブコードが生成される。ここで、SAVE<1>は、レジスタ割当て処理を行う前に S<1> に対して割当てられていたメモリ領域 (4 4, sp) を表している。

【 0 1 3 2 】

さらにステップ 3 2 0 5、3 2 0 3 (No) と処理は進み、ステップ 3 2 0 6 で $b=1$ に変更されて図 4 5 の処理は終了する。ステップ 2 8 0 5 の処理は、第 1 の実施の形態と同様の処理であるが、今の場合は J a v a バイトコードが imul であるので、図 2 4 に示したネイティブコードが生成される。ここではスタック深さ $js=3$ におけるオペランドは S<2>、S<3> であるが、それぞれレジスタ R10、R11 に割当てられているので、図 2 4 のオペランド割当てが両方ともレジスタの場合のネイティブコード、すなわち「mul S<2>, S<3>」というネイティブコードが生成される。以上が図 4 7 の (1 7) で示すネイティブコードが生成される処理である。

【 0 1 3 3 】

図 4 7 の (1 8) の変換処理において、ステップ 1 2 0 7 では図 4 2 に示す処理が行われる。ステップ 3 0 0 1 で求められる jsnext は 2 である。ステップ 3 0 0 2 では図 4 9 に示す処理が行われる。ステップ 3 3 0 1 では図 4 5 に示す処理が行われる ($top=b-1=0$ 、 $bottom=MAX(0, jsnext-NS)=0$)。ステップ 3 2 0 1、3 2 0 2、3 2 0 3 から $i=0$ となってステップ 3 2 0 4 が実行される。すなわちスタックオペランド [0] をメモリからレジスタに復帰するためのネイティブコードが生成される。今の例では、「ld S<0>, @SAVE<0>」というネイティブコー

ドが生成される。

【0134】

さらに制御はステップ3205、3203 (No) と進み、ステップ3206で $b=0$ に変更されて図45の処理は終了する。ステップ3003において「bra T28」というネイティブコードが生成される。図35の(8)で示すようにアドレス21に対してスタック深さ1が登録されているので、ステップ3004において求められる $jsnext$ は1である。ステップ3005で $b=MAX(0, jsnext-NS) = 0$ に変更される。以上が図47の(18)で示すネイティブコードが生成される処理である。

【0135】

以下同様にして、図47および図48に示すネイティブコードが生成される。

以上のように、本発明の第2の実施の形態による非ネイティブコードのプログラム実行方式によれば、ローカル変数とオペランドスタックとがレジスタに割当てられた結果、従来これらがメモリに割当てられていた場合に比べて、変換後のネイティブコード数が減少するとともに、メモリアクセス頻度も減少し、その結果実行速度も向上することになる。

【0136】

第3の実施の形態

本発明の第3の実施の形態に係るデータ処理装置の構成は多くの部分が第2の実施の形態の装置と共通である。

【0137】

まず図1～図18と図39とは、第3の実施の形態の装置で行なわれる処理は第2の実施の形態の装置で行なわれる処理と同じである。

【0138】

M32Rレジスタの使用方法は図43と同じである。

また図20の処理の流れ自体は同じであるが、ステップ1207における $jinst$ のJavaバイトコードに対するネイティブコードの生成処理は、図40～図42に示されるものではなく、対象とするJavaバイトコードの種類ごとに図50～図62のフローチャートに示す手順で行われる。

【 0 1 3 9 】

図 5 0 は、スタックへのデータ転送だけを行うような J a v a バイトコードに対する処理である。具体的な例では `iconst_<n>` と `iload_<n>` が相当する。この場合は、転送を行うネイティブコードを生成する代わりに、ステップ 3 6 0 1 において転送元の場所（あるいは、転送元が即値の場合にはその即値）を $P<i>$ に記録しておくだけである。ここで、 $P<i>$ ($i=0 \sim nStack-1$) は、即値か転送元の場所かを記録できるようなデータ構造である。 $P<i>$ は表 1 のいずれかの値をとる。以下の説明や図中では、 $P<i>$ の値とその意味とはそれぞれ下に示すような関係であるものとする。

【 0 1 4 0 】

【表 1】

記号	意味
x (即値)	スタックオペランド $[i]$ へ即値 x をロードするネイティブコードがペンディング中であることを示す。
$L<n>$	スタックオペランド $[i]$ にローカル変数 $[n]$ の値をロードするネイティブコードがペンディングであることを示す。
$S<i>$	スタックオペランド $[i]$ に割当てられたレジスタに有効な値が保持されていることを示す。
$SAVE<i>$	スタックオペランド $[i]$ には有効な値が保持されているが、割当てられたレジスタではなくメモリに格納されていることを示す。
-	特に意味のある記録がなされていないことを示す。

【 0 1 4 1 】

ステップ 3 6 0 2 は、実行の合流地点（すなわち分岐先となるラベルのあるアドレス）において、スタックオペランドのスタックトップ側の $NS(4)$ 個のエントリに割当てられたレジスタに値が保持されているようにするための処理である。スタックオペランドに割当てられたレジスタには必ずしも値を保持しているとは限らない。したがってそのままでは、複数の経路の実行パスが合流する地点において、レジスタの状態が不明となりそれ以降のネイティブコードの生成においてレジスタを利用することができなくなってしまう。そこで、実行の合流地点においては、必ずスタックオペランドのスタックトップ側の NS 個のスタックオペランドを割り当てられたレジスタに値を保持するようにしている。そのための

ネイティブコードは、次のような場合に挿入されることになる。

【0142】

実行が次のアドレスに継続するような J a v a バイトコードがラベルの直前にある場合（すなわち次のアドレスにラベルがある場合）、その J a v a バイトコードに対するネイティブコードの直後（ラベルの前）に挿入される（ステップ 3602、3702、3806、4003 による）。このネイティブコードの挿入処理は図 56 に示してある。

【0143】

分岐を行なうような J a v a バイトコードにおいて、分岐するネイティブコードを生成する直前に挿入される（ステップ 4101～4102、4203～4205、4208～4210、4215～4217 による）。

【0144】

このネイティブコードの挿入処理は図 51 および図 53 に示してある。

図 51 はスタックオペランドへの即値またはローカル変数の転送がペンディングになっている場合に、実際の転送を行なうネイティブコードを生成する処理である。ステップ 4401、4402、4411 の制御によって、 $i=0\sim k$ についてステップ 4403～4410 の処理を繰り返し行なう。すなわち、 $P\langle i \rangle$ が即値か $L\langle n \rangle$ である場合について、 $S\langle i \rangle$ レジスタをリザーブするための処理を行ない（ステップ 4404 と 4408）、次に $S\langle i \rangle$ レジスタに即値あるいは $L\langle n \rangle$ の値をロードするネイティブコードを生成している（4405、4409）。次に $S\langle i \rangle$ レジスタに有効値が保持されたことを示すため、 $P\langle i \rangle$ の値を $S\langle i \rangle$ に変更する（4406、4410）。

【0145】

$S\langle k \rangle$ レジスタをリザーブするためのネイティブコード生成処理（ステップ 3504 と 3707）は図 52 に示すフローチャートに従って行なわれる。まず $S\langle k \rangle$ レジスタがスタックオペランド $[k-NS]$ の値を保持しているかどうかを調べる（4602）。保持していなければ $S\langle k \rangle$ レジスタは空いている。保持しているようであれば、 $S\langle k \rangle$ レジスタ ($S\langle k-NS \rangle$ レジスタと同じ) の値を保存するために、「st S $\langle k-NS \rangle$ 、@SAVE $\langle k-NS \rangle$ 」というネイティブコードを生成する（4603）。次に S

<k-NS>レジスタの値が保存されたことを示すため、P<k-NS>の値をSAVE<k-NS>に変更する（4 6 0 4）。

【0 1 4 6】

図 5 3 はスタックオペランド [k] からスタックボトム側の NS 個のオペランドのうち、SAVE<i>に格納されているオペランドを、それらに対して割当てられたレジスタにロードするネイティブコードを生成する処理である。i=k-NS（ただし k<NS のときは 0）～k についてステップ 4 5 0 3 ～4 5 0 5 の処理を繰り返す行なう。すなわち、P<i>がSAVE<i>であれば（4 5 0 4）、「ld S<i>, @SAVE<i>」というネイティブコードを生成し（4 5 0 4）、S<i>レジスタに有効値が保持されたことを示すため、P<i>の値をS<i>に変更する（4 5 0 5）。

【0 1 4 7】

図 5 4 は iadd に対するネイティブコードの生成処理である。iadd はスタックオペランド [js-2] に書き込みを行なう（js は iadd 実行時のスタック深さ）ので、これに対して割当てられた S<js-2> のレジスタをリザーブするためのネイティブコードの生成処理をまず行なう（3 7 0 1）。次にステップ 3 7 0 2 では、P<js-2> と P<js-1> に記録された転送元のデータを加算して S<js-2> レジスタに格納するようなネイティブコードの生成を行なう。ここで生成するネイティブコードは P<js-2> と P<js-1> の値の組み合わせに応じて図 5 5 のようになる。特別な場合としてケース 1 と 6 のようにネイティブコードを生成しないこともある。これらの場合は P<js-2> には即値を記録しておくが、これ以外のケースでは S<js-2> に有効な値が保持されたことを示すため、P<js-2> を S<js-2> に変更する。ステップ 3 7 0 3 以降の処理は図 5 6 の通り（図 5 0 のステップ 3 6 0 2 と同じ）である。

【0 1 4 8】

図 5 6 は jpcnext での合流のためのネイティブコード挿入処理である。jpcnext にラベルがある場合、すなわちプログラム実行経路が jpcnext において合流する場合にステップ 4 3 0 2 ～4 3 0 4 の処理を行なう。

【0 1 4 9】

ステップ 4 3 0 2 では jpcnext におけるスタック深さの登録値を求めて jpcnext

の値とする。ステップ 4 3 0 3 では [0] ~ [jsnext-1] のスタックオペランドを有効化するためのネイティブコードの生成処理を行なう（この処理は図 5 1 の通りである）。ステップ 4 3 0 4 では S<jsnext-1> から下 N S 個のスタックオペランドをレジスタに保持するためのネイティブコードの生成処理を行なう。このステップの処理は図 5 3 の通りである。

【 0 1 5 0 】

i sub、i mul、i div についても i add の時の加算がそれぞれ減算、乗算、除算になる以外は i add と同様である。

【 0 1 5 1 】

図 5 7 は invokestatic<int (int,int)> に対するネイティブコードの生成処理である。invokestatic<int (int,int)> もリターン値を格納するために i add と同様にスタックオペランド [js-2] に書き込みを行なうので、i add と同様にこれに対して割当てられた S<js-2> のレジスタをリザーブするためのネイティブコードの生成処理をまず行なう（3 8 0 1）。次に P<js-2> と P<js-1> に記録された転送元のデータを引数としてスタックにプッシュするネイティブコードを生成する（3 8 0 2、3 8 0 3）。P<js-2> と P<js-1> の値に応じて生成されるネイティブコードは変化するが、i add のときと同様になるのでここでは図示していない。引き続き生成するネイティブコード（3 8 0 4、3 8 0 5）は、第 1 の実施の形態と同様である。ステップ 3 8 0 6 以降の処理は図 5 6 の通りである。

【 0 1 5 2 】

図 5 8 は ireturn に対するネイティブコードの生成処理である。まず P<js-1> に記録された転送元のデータを r 0 レジスタに転送するネイティブコードを生成する（3 9 0 1）。このとき生成するネイティブコードは、P<js-1> の値に応じて変化するが、ここでは図示していない。引き続き生成するネイティブコード 3 9 0 2 は、第 1 の実施の形態と同様である。最後に次のアドレス以降のネイティブコードの生成に備えて P<i> を設定している（3 9 0 3、3 9 0 4）。

【 0 1 5 3 】

図 5 9 は istore__<n> に対するネイティブコードの生成処理である。istore__<n> はローカル変数 [n] に書き込みを行なうが、ローカル変数への書き込みを行な

う前には、このローカル変数を転送元とするようなオペランドスタックへのデータ転送がペンディングになっている場合には、それらのデータ転送を実施しておく必要がある。そこでそのためのネイティブコードの生成をまず行なう（4 0 0 1）。次にP<js-1>に記録された転送元のデータをローカル変数[n]に転送するネイティブコードを生成する。このとき生成するネイティブコードはP<js-1>の値とL<n>がメモリかレジスタかによって変化するが、ここでは図示していない。ステップ4 0 0 3以降の処理は図5 6の通りである。

【0 1 5 4】

ステップ4 0 0 1における処理は図6 0に示してある。ステップ4 8 0 1、4 8 0 2および4 8 0 7を用いた制御によって、i=0～js-1についてステップ4 8 0 3～4 8 0 6の処理を繰り返し行なう。すなわち、P<i>にL<n>が記録されているならば（4 8 0 3）、S<i>レジスタをリザーブし（4 8 0 4）、S<i>レジスタにL<n>をロードするネイティブコードを生成し（4 8 0 4）、S<i>に有効値が保持されたことを示すため、P<i>をS<i>に変更する（4 8 0 6）。

【0 1 5 5】

図6 1はgotoに対するネイティブコードの生成処理である。分岐のネイティブコード（bra）を生成する（4 1 0 3）が、その前に、スタックオペランド[js-1]からスタックトップ側のNS個のエントリに割当てられたレジスタに値を保持するためのネイティブコードを生成する（4 1 0 1、4 1 0 2）。これらの処理は図5 3、図5 2の通りである。

【0 1 5 6】

図6 2はifgeに対するネイティブコードの生成処理である。P<js-1>に記録された転送元のデータと0を比較するネイティブコードと、その転送元のデータ>=0であるなら分岐するような条件分岐のネイティブコードとを生成するが、その前にスタックトップのNS個のエントリに割当てられたレジスタに値を保持するためのネイティブコードを生成する。なおここでのスタックトップはifgeの実行後のスタックトップであるのでk=js-2である（ステップ4 2 0 3～4 2 0 5、4 2 0 8～4 2 1 0、4 2 1 5～4 2 1 7）。

【0 1 5 7】

以下、図 6 で示した J a v a バイトコードのメソッドを図 1 4 の処理手順に従ってネイティブコードのサブルーチンに変換する例について説明する。

【0 1 5 8】

ステップ 6 0 2 までは第 2 の実施の形態と同様に行なわれる。すなわち、図 4 6 に示すようなレジスタ割当て結果となる。

【0 1 5 9】

ステップ 6 0 3 では図 2 0 に示す処理が行なわれる。図 6 3 はこの処理を実行するにつれて生成されるネイティブコードを示したものである。

【0 1 6 0】

まず、ステップ 1 2 0 1 では、図 6 3 の状態 (1) で示すようなネイティブコードが生成される。これは、図 3 2 から図 3 3 ヘスタックレイアウトを変更するためのネイティブコードと、ローカル変数のうち、引数であってレジスタに割当てられているもの (この例では L<0> と L<2>) についてメモリからレジスタに値を読み込んでおくためのネイティブコードである。

【0 1 6 1】

ステップ 1 2 0 2 において jpc=0 となる。ステップ 1 2 0 3 において求められるアドレス 0 にある J a v a バイトコード jinst は iload__0 である。ステップ 1 2 0 4 において求められる jinstsize は 1 である。ステップ 1 2 0 5 において求められる jpcnext は 1 である。図 3 5 の (1) に示すようにアドレス 0 に対応付けてスタック深さ 0 が登録されているため、ステップ 1 2 0 6 において求められる js は 0 となる。

【0 1 6 2】

ステップ 1 2 0 7 では変換しようとする J a v a バイトコード jinst に応じて図 5 0 ～図 6 2 に示すような処理が行なわれる。今の例の場合は、jinst は iload__0 であるので、図 5 0 に示す処理が行なわれる。ステップ 4 0 0 1 において P<0> に L<0> が記録される。ステップ 3 6 0 2 から No に進んで図 5 0 の処理は終了する。以上が図 6 3 の (2) の状態である。結局 iload__0 に対するネイティブコードはこの (2) では生成されないことになる。

【0 1 6 3】

図 6 3 の (3) でも同様に P<1>に L<1>が記録されるだけでネイティブコードは生成されない。

【 0 1 6 4 】

(4) ではステップ 1 2 0 7 において図 5 4 の処理が行なわれる。ステップ 3 7 0 1 では図 5 2 に示す処理が行なわれる ($i = js - 2 = 0$) が、ステップ 4 6 0 1 における判定結果が「NO」となり処理は終了する。ステップ 3 7 0 2 でまず図 5 5 のネイティブコードが生成される。今の場合、スタック深さ $js = 2$ におけるオペランド P<0>と P<1>とはそれぞれ L<0> (レジスタ) 、 L<1> (レジスタ) であるので、ケース番号 2 9 のネイティブコードが生成される。P<0>は S<0>に変更されて図 5 2 の処理が終了する。ステップ 3 7 0 3 では図 5 6 の処理が行なわれるが、ステップ 4 3 0 1 における判定結果が「NO」となり処理を終了する。以上で図 5 4 の処理を終了する。

【 0 1 6 5 】

図 6 3 の (5) の変換処理においては、ステップ 1 2 0 7 で図 5 9 の処理が行なわれる。図 5 9 のステップ 4 0 0 1 では図 6 0 に示す処理が行なわれる。今の場合 L<2>が登録されている P<i>はないので、ステップ 4 8 0 3 における判定結果は常に「NO」となり、ネイティブコードは生成されずに終了する。スタック深さ $js = 1$ において P<0>は「 S<0>」となっているので、ステップ 4 0 0 2 では S<0>をローカル変数 [2] に格納するようなネイティブコードを生成する。図 4 6 に示されるようにローカル変数 [2] はメモリに割当てられているので、「st S<0>, @L<2>」というネイティブコードが生成されることになる。ステップ 4 0 0 3 では図 5 6 の処理が行なわれるが、ステップ 4 3 0 1 における判定結果が NO となって処理を終了する。以上で図 5 9 の処理を終了する。

【 0 1 6 6 】

図 6 3 の (6) に対する変換処理は (2) の場合と同様であり、P<0>に即値 1 が記録され、ネイティブコードは生成されない。

【 0 1 6 7 】

(7) も同様に P<1>に L<0>が記録され、ネイティブコードは生成されない。

(8) ではステップ 1 2 0 7 において図 6 2 の処理が行なわれる。今の場合ス

タック深さ $js=2$ において $P<1>$ は $L<0>$ となっているので、制御はステップ 4 2 0 1、4 2 0 7、4 2 0 8 と進む。ステップ 4 2 0 8 で求められる $jsnext$ は 1 である。ステップ 4 2 0 9 では図 5 1 に示す処理が行なわれる ($k=0$)。ステップ 4 4 0 1、4 4 0 2 から $i=0$ となって制御はステップ 4 4 0 3 に進む。 $P<0>$ は即値なので、制御はステップ 4 4 0 4 に進む。ステップ 4 4 0 4 では図 5 2 に示す処理が行なわれる ($k=0$) が、ステップ 4 6 0 1 から No の側に進んで図 5 2 の処理は終了する。ステップ 4 4 0 5 において $S<0>$ に $P<0>$ の即値 (1) をロードするネイティブコードが生成される。ここでは「ldi $S<0>$, #1」というネイティブコードが生成されることになる。ステップ 4 4 0 6 で $P<0>$ は $S<0>$ に書き換えられる。ステップ 4 4 0 9、4 4 0 2 から No に進んで図 5 1 の処理は終了する。

【0168】

図 6 2 のステップ 4 2 0 5 では図 5 3 に示す処理が行なわれる ($k=0$) が、この場合、 $SAVE<i>$ が記録されている $P<i>$ は存在しないので、ステップ 4 5 0 3 での判定結果が「NO」となり、ネイティブコードを生成することなく図 5 3 の処理は終了する。ステップ 4 2 0 6 で「bra T21」というネイティブコードが生成される。

【0169】

図 6 3 の (9) ~ (13) は、それぞれ (6) (2) (2) (6) (2) と同様に処理され、 $P<1>$ ~ $P<5>$ にそれぞれ即値 2、 $L<0>$ 、 $L<1>$ 、即値 3、 $L<2>$ が記録される。

【0170】

図 6 3 の (14) ではステップ 1 2 0 7 において図 5 4 の処理が行なわれる。図 5 4 のステップ 3 7 0 1 では図 5 2 に示す処理が行なわれる ($k=js-2=4$)。ここではステップ 4 6 0 1 (Yes)、4 6 0 2 と進む。 $P<0>$ には $SAVE<0>$ が記録されているので、ステップ 4 6 0 3 に進む。ステップ 4 6 0 3 では $S<0>$ を保存するネイティブコードが生成される。すなわち、「st $S<0>$, @SAVE<0>」というネイティブコードが生成される。ステップ 4 6 0 4 において $P<0>$ が $SAVE<0>$ に変更されて図 5 2 の処理が終了する。

【0171】

図54のステップ3702では図55に示すネイティブコードが生成される。今の場合、スタック深さ $js=6$ におけるオペランド $P<4>$ と $P<5>$ はそれぞれ即値3、 $L<2>$ （メモリ）であるので、ケース番号5のネイティブコードが生成される。 $P<4>$ は $S<4>$ に変更されて図56の処理を終了し、図54の処理を終了する。

【0172】

図63の(15)ではステップ1207において図54の処理が行なわれる（加算の代わりに減算となる）。ステップ3701では図52に示す処理が行なわれる（ $k=js-2=3$ ）が、ここではステップ4601の判定結果が「NO」となり処理を終了する。ステップ3702では図55と同様のネイティブコードが生成される。今の場合、スタック深さ $js=5$ におけるオペランド $P<3>$ と $P<4>$ はそれぞれ $L<1>$ （レジスタ）、 $S<4>$ （レジスタ）であるので、ケース番号27のネイティブコードが生成される（ただし、`add`ではなく`div`を使用する）。 $P<3>$ は $S<3>$ に変更される。

【0173】

ステップ3703では図56の処理が行なわれるが、ステップ4301の判定結果が「NO」となり処理を終了する。以上で図54の処理を終了する。

【0174】

図63の(16)の変換処理では、ステップ1207において図54の処理が行なわれる。図54のステップ3701では図52に示す処理が行なわれる（ $k=js-2=2$ ）が、ここではステップ4601での判定結果が「NO」となり処理を終了する。ステップ3702では図55のネイティブコードが生成される。今の場合、スタック深さ $js=4$ におけるオペランド $P<2>$ と $P<3>$ はそれぞれ $L<0>$ （レジスタ）、 $S<3>$ （レジスタ）であるので、ケース番号27のネイティブコードが生成される。 $P<2>$ は $S<2>$ に変更される。ステップ3703では図56の処理が行なわれるが、ステップ4301での判定結果が「NO」となり処理を終了する。以上で図54の処理を終了する。

【0175】

図63の(17)の変換処理では、ステップ1207において図54の処理が行なわれる（ただし加算でなく乗算である）。ステップ3701では図52に示

す処理が行なわれる ($k = js - 2 = 1$) が、ここではステップ 4 6 0 1 での判定結果が「NO」となり、図 5 2 の処理を終了する。

【0 1 7 6】

図 5 4 に戻り、ステップ 3 7 0 2 では図 5 5 のネイティブコードが生成される (ただし、add ではなく mul を使用する)。今の場合、スタック深さ $js = 3$ におけるオペランド $P<1>$ と $P<2>$ はそれぞれ即値 2、 $S<2>$ (レジスタ) であるので、ケース番号 2 のネイティブコードが生成される。即値 2 の掛け算は 2 ビット左シフトで求められるので、mul 命令でなく `sl 3` 命令が生成される。 $P<1>$ は $S<1>$ に変更される。ステップ 3 7 0 3 では図 5 6 の処理が行なわれるが、ステップ 4 3 0 1 での判定結果が「NO」となり、何もせず図 5 6 の処理を終了する。以上で図 5 4 の処理を終了する。

【0 1 7 7】

図 6 3 の (18) の変換処理では、ステップ 1 2 0 7 において図 6 1 の処理が行なわれる。図 6 1 のステップ 4 1 0 1 では図 5 1 に示す処理が行なわれる ($k = js - 1 = 1$) が、今の場合、即値またはローカル変数が記録されている $P<i>$ は存在しないため、 $i = 0, 1$ の場合ともにステップ 4 4 0 3 (NO)、4 4 0 7 (NO) と進行して図 5 1 の処理を終了する。

【0 1 7 8】

図 6 1 に戻り、ステップ 4 1 0 2 では図 5 3 の処理が行なわれる ($k = js - 1 = 1$)。ステップ 4 5 0 1 で求められる i は 0 である。ステップ 4 5 0 2、4 5 0 3 と進み、 $P<0>$ に `SAVE<0>` が記録されているのでステップ 4 5 0 4 に進む。ステップ 4 5 0 4 では `SAVE<0>` のメモリ領域の値をレジスタ $S<0>$ にロードするネイティブコードが生成される。すなわち `ld S<0>, @SAVE<0>` というネイティブコードが生成される。ステップ 4 5 0 5 では $P<0>$ が $S<0>$ に変更される。ステップ 4 5 0 6 で $i = 1$ となってステップ 4 5 0 2、4 5 0 3 と進む。 $P<1>$ は $S<1>$ となっているので、ステップ 4 5 0 3 での判定結果は「NO」となる。したがってステップ 4 5 0 6 で $i = 2$ となり、ステップ 4 5 0 2 での判定結果が「NO」となり図 5 3 の処理を終了する。

【0 1 7 9】

図 6 1 に戻り、ステップ 4 1 0 3 では「bra T28」というネイティブコードが生成される。以上が図 6 3 の (18) の状態である。

【0180】

図 6 1 のステップ 4 1 0 4 で求められる jsnext は 1 であり、ステップ 4 1 0 5 では P<0> が S<0> になる。以上が図 6 3 の (18') である。

【0181】

図 6 4 の (19) の変換処理では (2) と同様にして P<1> に L<0> が記録され、ネイティブコードは生成されない。

【0182】

(20) では (6) と同様にして P<2> に 1 が記録され、ネイティブコードは生成されない。

【0183】

(21) では (4) と同様にして図のネイティブコードが生成される (ただし加算でなく減算となる)。この場合は図 5 5 のケース番号 = 25 に相当する。P<1> が S<1> に変更される。

【0184】

(22) では (2) と同様にして P<2> に L<2> が記録され、ネイティブコードは生成されない。

【0185】

(23) ではステップ 1 2 0 7 において図 5 7 の処理が行なわれる。図 5 7 のステップ 3 8 0 1 では図 5 2 の処理が行なわれる ($k = js - 2 = 1$) が、ステップ 4 6 0 1 での判定結果が「NO」となって処理を終了し図 5 7 のステップ 3 8 0 2 に戻る。P<1> は S<1> であるのでステップ 3 8 0 2 で生成されるネイティブコードは push S<1> となる。P<2> は L<2> であり、L<2> はメモリに割当てられているので、図 5 7 のステップ 3 8 0 3 で生成されるネイティブコードは「ld r0, @L<2>」と「push r0」とになる。ステップ 3 8 0 4、3 8 0 5 と進んで図に示すネイティブコードが生成される。ステップ 3 8 0 6 では図 5 6 の処理が行なわれるが、ネイティブコードは生成されない。

【0186】

図 6 4 の (2 4) の変換処理では、(4) と同様にして図のネイティブコードが生成される。この場合は図 5 5 のケース番号 1 4 に相当する。P<0>がS<0>に変更される。

【 0 1 8 7 】

(2 5) の変換処理では、ステップ 1 2 0 7 において図 5 8 の処理が行われる。P<0>はS<0>であるので、ステップ 3 9 0 1 で生成されるネイティブコードはmv r0, S<0>となる。ステップ 3 9 0 2 に進んで図 6 4 の (2 5) のネイティブコードが生成される。

【 0 1 8 8 】

以上のようにこの第 3 の実施の形態のデータ処理装置によれば、データ転送だけを行なう J a v a バイトコードについてはネイティブコードを生成せず、転送先のオペランドに転送元を対応付けて記録しておく。演算を行なう J a v a バイトコードを変換する際に、先に転送元が記録されているようなオペランドを使用する際には記録しておいた転送元を使用して演算を行うようなネイティブコードを生成する。そのため、変換後のネイティブコード数が減少し、その結果実行速度も向上することになる。

【 0 1 8 9 】

今回開示された実施の形態はすべての点で例示であって制限的なものではないと考えられるべきである。本発明の範囲は上記した説明ではなくて特許請求の範囲によって示され、特許請求の範囲と均等の意味および範囲内でのすべての変更が含まれることが意図される。

【 0 1 9 0 】

【発明の効果】

請求項 1 に記載の発明によれば、非ネイティブコードを、比較的少ない容量の記憶手段を用いて、高速に実行できる。

【 0 1 9 1 】

請求項 2 に記載の発明によれば、ハードウェア量をより少なく、および／または全体の処理速度をより高くすることができる。

【 0 1 9 2 】

請求項3に記載の発明によれば、ハードウェア量をより少なく、および／または全体の処理速度をより高くすることができる。

【0193】

請求項4に記載の発明によれば、変換後のプログラムの実行を高速化できる。

請求項5に記載の発明によれば、変換後のプログラムの実行を高速化できる。

【0194】

請求項6に記載の発明にかかるデータ処理装置は、請求項5に記載の発明の構成に加えて、コード変換手段は、スタックへのデータの転送のみを行なう非ネイティブコードを検出してその転送元と転送先とを対応付けて記憶するための手段をさらに含み、生成するための手段は、データの転送のみを行なう非ネイティブコードの検出メモリオペランドのうちのスタックオペランドのうち、スタックトップ側のスタックオペランドが、プロセッサに備えられたレジスタに保持されるように、データ処理装置に接続されるメモリと、プロセッサのレジスタとの間で実行時にデータの保存と復帰とを行なうように、かつ、転送先をオペランドとして使用するコードについては、転送先に代えて、記憶するための手段に記憶された転送元をオペランドとするように、ネイティブコードを生成するための手段を含む。

【0195】

変換後のネイティブコードの数が、このような処理を行なわない場合と比較して減少する。実行すべきコード数が減少するので、変換後のプログラムの実行を高速化できる。

【図面の簡単な説明】

【図1】 第1～第3の実施の形態におけるデータ処理装置を示す図である。

【図2】 第1～第3の実施の形態における演算部の命令セットを示す図である。

【図3】 第1～第3の実施の形態における演算部の命令セットを示す図である。

【図4】 第1～第3の実施の形態における演算部の命令セットを示す図で

ある。

【図 5】 多機能命令デコーダ 1 0 5 のブロック図である。

【図 6】 第 1 ～第 3 の実施の形態の説明中、実行例で使用する変換対象の J a v a バイトコードで記述されたメソッドと J a v a バイトコードの意味を示す図である。

【図 7】 第 1 ～第 3 の実施の形態における非ネイティブコードのプログラム実行処理の手順を示すフローチャートである。

【図 8】 第 1 ～第 3 の実施の形態における非ネイティブコードのサブルーチン呼出し処理の手順を示すフローチャートである。

【図 9】 第 1 ～第 3 の実施の形態における実行方法選択処理の手順を示すフローチャートである。

【図 1 0】 第 1 ～第 3 の実施の形態における非ネイティブまたはネイティブコードの実行処理の手順を示すフローチャートである。

【図 1 1】 第 1 ～第 3 の実施の形態における複雑な処理を要する非ネイティブコードをネイティブコードに変換する処理の手順を示すフローチャートである。

【図 1 2】 第 1 ～第 3 の実施の形態における `invokestatic` をネイティブコードに変換する処理の手順を示すフローチャートである。

【図 1 3】 第 1 ～第 3 の実施の形態における実行前準備処理の手順を示すフローチャートである。

【図 1 4】 第 1 ～第 3 の実施の形態における非ネイティブコードのサブルーチンをネイティブコードのサブルーチンに変換する処理の手順を示すフローチャートである。

【図 1 5】 第 1 ～第 3 の実施の形態における変換開始処理の手順を示すフローチャートである。

【図 1 6】 第 1 ～第 3 の実施の形態におけるレジスタ割当て処理の手順を示すフローチャートである。

【図 1 7】 第 1 ～第 3 の実施の形態におけるアドレスとスタック深さの登録処理の手順を示すフローチャートである。

【図 18】 第 1～第 3 の実施の形態における jinst のオペランド参照のカウント処理の手順を示すフローチャートである。

【図 19】 第 1 の実施の形態におけるレジスタ割当ての決定処理の手順を示すフローチャートである。

【図 20】 第 1～第 3 の実施の形態におけるメソッドコード変換処理の手順を示すフローチャートである。

【図 21】 第 1 の実施の形態、3 における iconst_<n> に対して生成されるネイティブコードを示す図である。

【図 22】 第 1 の実施の形態、3 における iload_<n> に対して生成されるネイティブコードを示す図である。

【図 23】 第 1 の実施の形態、3 における istore_<n> に対して生成されるネイティブコードを示す図である。

【図 24】 第 1 の実施の形態、3 における iadd に対して生成されるネイティブコードを示す図である。

【図 25】 第 1 の実施の形態における ifge に対して生成されるネイティブコードを示す図である。

【図 26】 第 1 の実施の形態における goto に対して生成されるネイティブコードを示す図である。

【図 27】 第 1 の実施の形態における ireturn に対して生成されるネイティブコードを示す図である。

【図 28】 第 1 の実施の形態における invokestatic に対して生成されるネイティブコードを示す図である。

【図 29】 第 1 の実施の形態における M32R のレジスタ使用方法を示す図である。

【図 30】 第 1～第 3 の実施の形態におけるスタック使用方法を示す図である。

【図 31】 第 1～第 3 の実施の形態におけるスタック使用方法を示す図である。

【図 32】 第 1～第 3 の実施の形態におけるスタック使用方法を示す図で

ある。

【図 3 3】 第 1 ～ 第 3 の実施の形態におけるスタック使用方法を示す図である。

【図 3 4】 第 1 の実施の形態におけるオペランドへのレジスタ割当ての状態を示す図である。

【図 3 5】 第 1 の実施の形態におけるレジスタ割当て処理の実行例を示す図である。

【図 3 6】 第 1 の実施の形態におけるレジスタ割当て処理の実行例を示す図である。

【図 3 7】 第 1 の実施の形態におけるメソッドコード変換処理の実行例を示す図である。

【図 3 8】 第 1 の実施の形態におけるメソッドコード変換処理の実行例を示す図である。

【図 3 9】 第 2 の実施の形態におけるレジスタ割当ての決定処理の手順を示すフローチャートである。

【図 4 0】 第 2 の実施の形態における分岐しない J a v a バイトコードに対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 4 1】 第 2 の実施の形態における if ge に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 4 2】 第 2 の実施の形態における goto に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 4 3】 第 2 の実施の形態における M 3 2 R レジスタの使用法を示す図である。

【図 4 4】 第 2 の実施の形態における S<p>～S<q>レジスタのオーバーフロー対処のためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 4 5】 第 2 の実施の形態における S<bottom>～S<top>レジスタのアンダフロー対処のためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 4 6】 第 2 の実施の形態におけるオペランドへのレジスタ割当ての状態を示す図である。

【図 4 7】 第 2 の実施の形態におけるメソッドコード変換処理の実行例を示す図である。

【図 4 8】 第 2 の実施の形態におけるメソッドコード変換処理の実行例を示す図である。

【図 4 9】 第 2 の実施の形態における $S\langle jnext-1 \rangle$ から下の NS 個のスタックオペランドをレジスタに保持するためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 0】 第 3 の実施の形態におけるスタックへのデータ転送だけを行なう `Java` バイトコードに対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 1】 第 3 の実施の形態における $P\langle i \rangle$ を有効化するためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 2】 第 3 の実施の形態における $S\langle k \rangle$ レジスタをリザーブするためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 3】 第 3 の実施の形態における $S\langle k \rangle$ から下の NS 個のスタックオペランドをレジスタに保持するためのネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 4】 第 3 の実施の形態における `iadd` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 5】 第 3 の実施の形態における `iadd` に対して生成されるネイティブコードを示す図である。

【図 5 6】 第 3 の実施の形態における `jpcnext` での合流のためのネイティブコード生成処理を示す図である。

【図 5 7】 第 3 の実施の形態における `invokestatic` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 8】 第 3 の実施の形態における `ireturn` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 5 9】 第 3 の実施の形態における `istore__<n>` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 6 0】 第 3 の実施の形態における `L<n>` のパージ処理を示す図である。

【図 6 1】 第 3 の実施の形態における `goto` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 6 2】 第 3 の実施の形態における `ifge` に対するネイティブコードの生成処理の手順を示すフローチャートである。

【図 6 3】 第 3 の実施の形態のデータ処理装置におけるメソッドコード変換処理の実行例を示す図である。

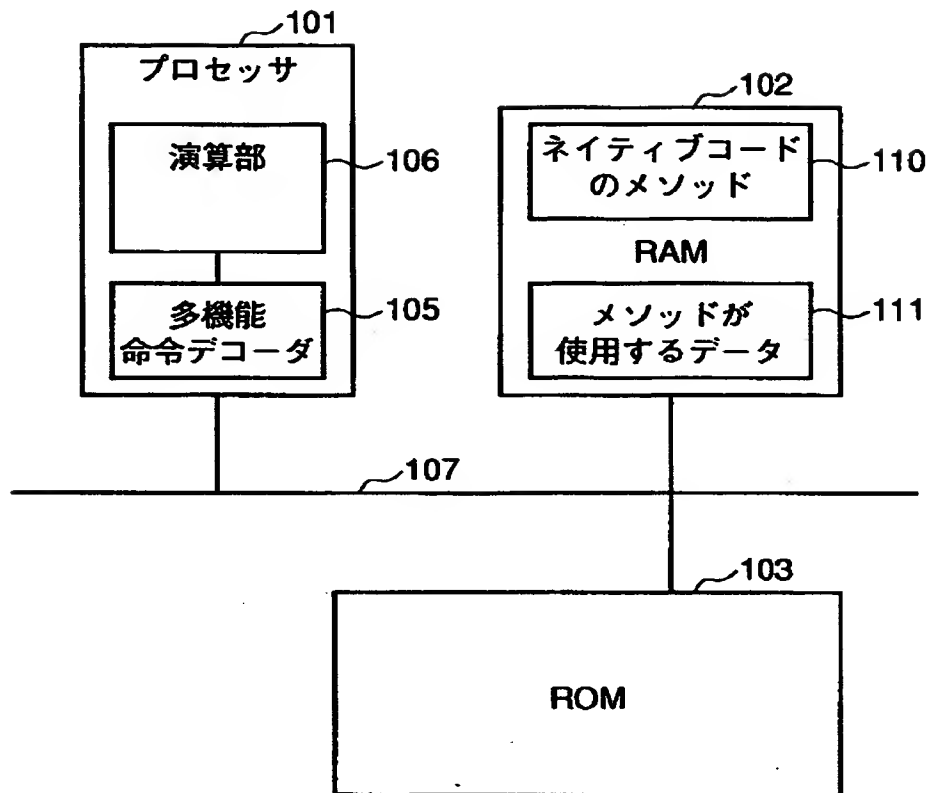
【図 6 4】 第 3 の実施の形態のデータ処理装置におけるメソッドコード変換処理の実行例を示す図である。

【符号の説明】

1 0 1 プロセッサ、1 0 2 RAM、1 0 3 ROM、1 0 5 多機能命令デコーダ、1 0 6 演算部、1 0 7 バス、1 0 7 A データバス、1 0 7 B アドレスバス、1 1 0 ネイティブコードのメソッド、1 1 1 メソッドが使用するデータ、1 2 0 ハードウェアトランスレータ、1 2 1 マルチプレクサ、1 2 2 比較回路。

【書類名】 図面

【図1】



【図2】

ニーモニック	動作	条件ビット(C)
ADD Rdest, Rsrc	Rdest = Rdest + Rsrc	—
ADD3 Rdest, Rsrc, #imm16	Rdest = Rsrc + (sh)imm16	—
ADDI Rdest, #imm8	Rdest = Rdest + (sh)imm8	—
ADDV Rdest, Rsrc	Rdest = Rdest + Rsrc	変化
ADDV3 Rdest, Rsrc, #imm16	Rdest = Rsrc + (sh)imm16	変化
ADDX Rdest, Rsrc	Rdest = Rdest + Rsrc + C	変化
AND Rdest, Rsrc	Rdest = Rdest & Rsrc	—
AND3 Rdest, Rsrc, #imm16	Rdest = Rsrc & (uh)imm16	—
BC pcdisp8	if(C) PC=PC+((sh)pcdisp8<<2)	—
BC pcdisp24	if(C) PC=PC+((s24)pcdisp24<<2)	—
BEQ Rsrc1, Rsrc2, pcdisp16	if(Rsrc1 == Rsrc2) PC=PC+((sh)pcdisp16<<2)	—
BEQZ Rsrc, pcdisp16	if(Rsrc == 0) PC=PC+((sh)pcdisp16<<2)	—
BGEZ Rsrc, pcdisp16	if(Rsrc >= 0) PC=PC+((sh)pcdisp16<<2)	—
BGTZ Rsrc, pcdisp16	if(Rsrc > 0) PC=PC+((sh)pcdisp16<<2)	—
BL pcdisp8	R14=PC+4, PC=PC+((sh)pcdisp8<<2)	—
BL pcdisp24	R14=PC+4, PC=PC+((s24)pcdisp24<<2)	—
BLEZ Rsrc, pcdisp16	if(Rsrc <= 0) PC=PC+((sh)pcdisp16<<2)	—
BLTZ Rsrc, pcdisp16	if(Rsrc < 0) PC=PC+((sh)pcdisp16<<2)	—
BNC pcdisp8	if(!C) PC=PC+((sh)pcdisp8<<2)	—
BNC pcdisp24	if(!C) PC=PC+((s24)pcdisp24<<2)	—
BNE Rsrc1, Rsrc2, pcdisp16	if(Rsrc1 != Rsrc2) PC=PC+((sh)pcdisp16<<2)	—
BNEZ Rsrc, pcdisp16	if(Rsrc != 0) PC=PC+((sh)pcdisp16<<2)	—
BRA pcdisp8	PC=PC+((sh)pcdisp8<<2)	—
BRA pcdisp24	PC=PC+((s24)pcdisp24<<2)	—
CMP Rsrc1, Rsrc2	(s)Rsrc1 < (s)Rsrc2	変化
CMPI Rsrc, #imm16	(s)Rsrc < (sh)imm16	変化
CMPU Rsrc1, Rsrc2	(u)Rsrc1 < (u)Rsrc2	変化
CMPI Rsrc, #imm16	(u)Rsrc < (u)((sh)imm16)	変化
DIV Rdest, Rsrc	Rdest = (s)Rdest / (s)Rsrc	—
DIVU Rdest, Rsrc	Rdest = (u)Rdest / (u)Rsrc	—
JL Rsrc	R14 = PC+4, PC = Rsrc	—
JMP Rsrc	PC = Rsrc	—
LD Rdest, @(disp16, Rsrc)	Rdest = *(s*)(Rsrc+(sh)disp16)	—
LD Rdest, @Rsrc	Rdest = *(s*)Rsrc	—
LD Rdest, @Rsrc+	Rdest = *(s*)Rsrc, Rsrc += 4	—

【図 3】

ニーモニック	動作	条件ビット(C)	
LD24	Rdest, #imm24	Rdest = imm24 & 0x00ffffff	—
LDB	Rdest, @(disp16, Rsrc)	Rdest = *(sb *) (Rsrc+(sh)disp16)	—
LDB	Rdest, @Rsrc	Rdest = *(sb *) Rsrc	—
LDH	Rdest, @(disp16, Rsrc)	Rdest = *(sh *) (Rsrc+(sh)disp16)	—
LDH	Rdest, @Rsrc	Rdest = *(sh *) Rsrc	—
LDI	Rdest, #imm16	Rdest = (sh)imm16	—
LDI	Rdest, #imm8	Rdest = (sb)imm8	—
LDUB	Rdest, @(disp16, Rsrc)	Rdest = *(ub *) (Rsrc+(sh)disp16)	—
LDUB	Rdest, @Rsrc	Rdest = *(ub *) Rsrc	—
LDUH	Rdest, @(disp16, Rsrc)	Rdest = *(uh *) (Rsrc+(sh)disp16)	—
LDUH	Rdest, @Rsrc	Rdest = *(uh *) Rsrc	—
LOCK	Rdest, @Rsrc	LOCK = 1, Rdest = *(s *) Rsrc	—
MACHI	Rsrc1, Rsrc2	accumulator += (s)(Rsrc1 & 0xffff0000), 	

【図 4】

ニーモニック	動作	条件ビット(C)
SETH Rdest, #imm16	Rdest = imm16 << 16	—
SLL Rdest, Rsrc	Rdest = Rdest << (Rsrc & 31)	—
SLL3 Rdest, Rsrc, #imm16	Rdest = Rsrc << (imm16 & 31)	—
SLLI Rdest, #imm5	Rdest = Rdest << imm5	—
SRA Rdest, Rsrc	Rdest = (s)Rdest >> (Rsrc & 31)	—
SRA3 Rdest, Rsrc, #imm16	Rdest = (s)Rsrc >> (imm16 & 31)	—
SRAI Rdest, #imm5	Rdest = (s)Rdest >> imm5	—
SEL Rdest, Rsrc	Rdest = (u)Rdest >> (Rsrc & 31)	—
SEL3 Rdest, Rsrc, #imm16	Rdest = (u)Rsrc >> (imm16 & 31)	—
SELI Rdest, #imm5	Rdest = (u)Rdest >> imm5	—
ST Rsrc1, @(displ6, Rsrc2)	*(s *) (Rsrc2 + (sh)displ6) = Rsrc1	—
ST Rsrc1, @+Rsrc2	Rsrc2 += 4, *(s *)Rsrc2 = Rsrc1	—
ST Rsrc1, @-Rsrc2	Rsrc2 -= 4, *(s *)Rsrc2 = Rsrc1	—
ST Rsrc1, @Rsrc2	*(s *)Rsrc2 = Rsrc1	—
STB Rsrc1, @(displ6, Rsrc2)	*(sb *) (Rsrc2 + (sh)displ6) = Rsrc1	—
STB Rsrc1, @Rsrc2	*(sb *)Rsrc2 = Rsrc1	—
STH Rsrc1, @(displ6, Rsrc2)	*(sh *) (Rsrc2 + (sh)displ6) = Rsrc1	—
STH Rsrc1, @Rsrc2	*(sh *)Rsrc2 = Rsrc1	—
SUB Rdest, Rsrc	Rdest = Rdest - Rsrc	—
SUBV Rdest, Rsrc	Rdest = Rdest - Rsrc	変化
SUBX Rdest, Rsrc	Rdest = Rdest - Rsrc - C	変化
TRAP #n	PSW[BSM, BIE, BC] = PSW[SM, IE, C] PSW[SM, IE, C] = PSW[SM, 0, 0] Call trap-handler number-n	変化
UNLOCK Rsrc1, @Rsrc2	if(LOCK) (*(s *)Rsrc2 = Rsrc1,) LOCK=0	—
XOR Rdest, Rsrc	Rdest = Rdest ^ Rsrc	—
XOR3 Rdest, Rsrc, #imm16	Rdest = Rsrc ^ (uh)imm16	—

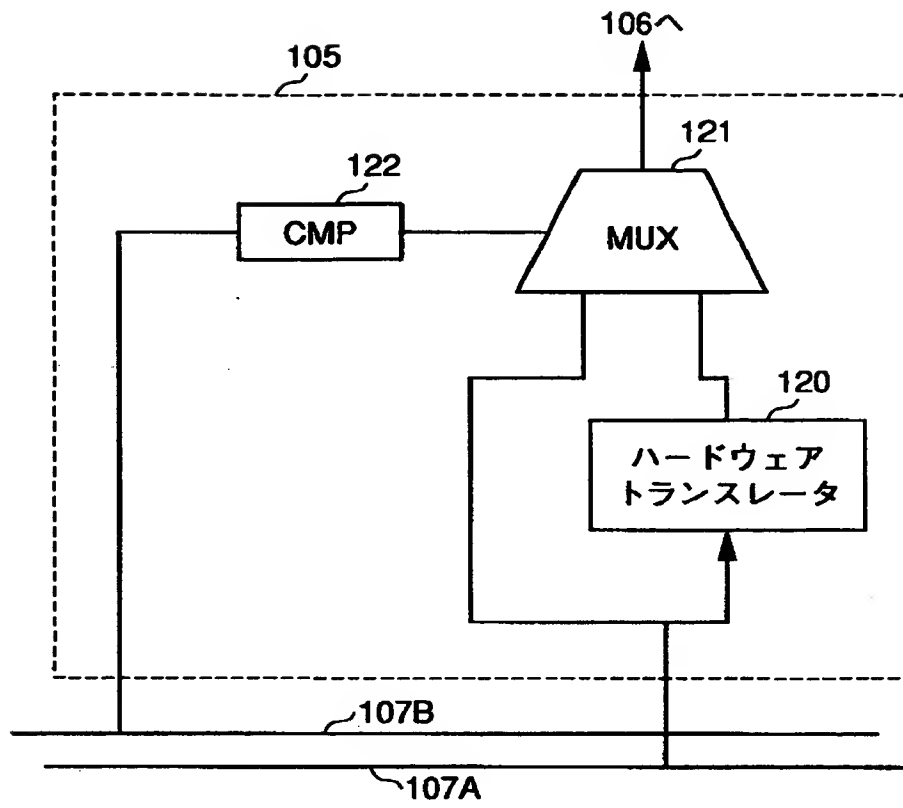
Where:

```

typedef signed int    s; /* 32 bit signed integer (word)*/
typedef unsigned int  u; /* 32 bit unsigned integer (word)*/
typedef signed short  sh; /* 16 bit signed integer (halfword)*/
typedef unsigned short uh; /* 16 bit unsigned integer (halfword)*/
typedef signed char   sb; /* 8 bit signed integer (byte)*/
typedef unsigned char ub; /* 8 bit unsigned integer (byte)*/

```

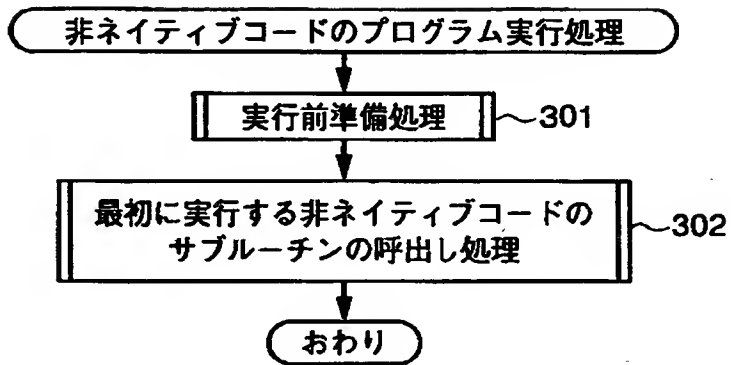
【図 5】



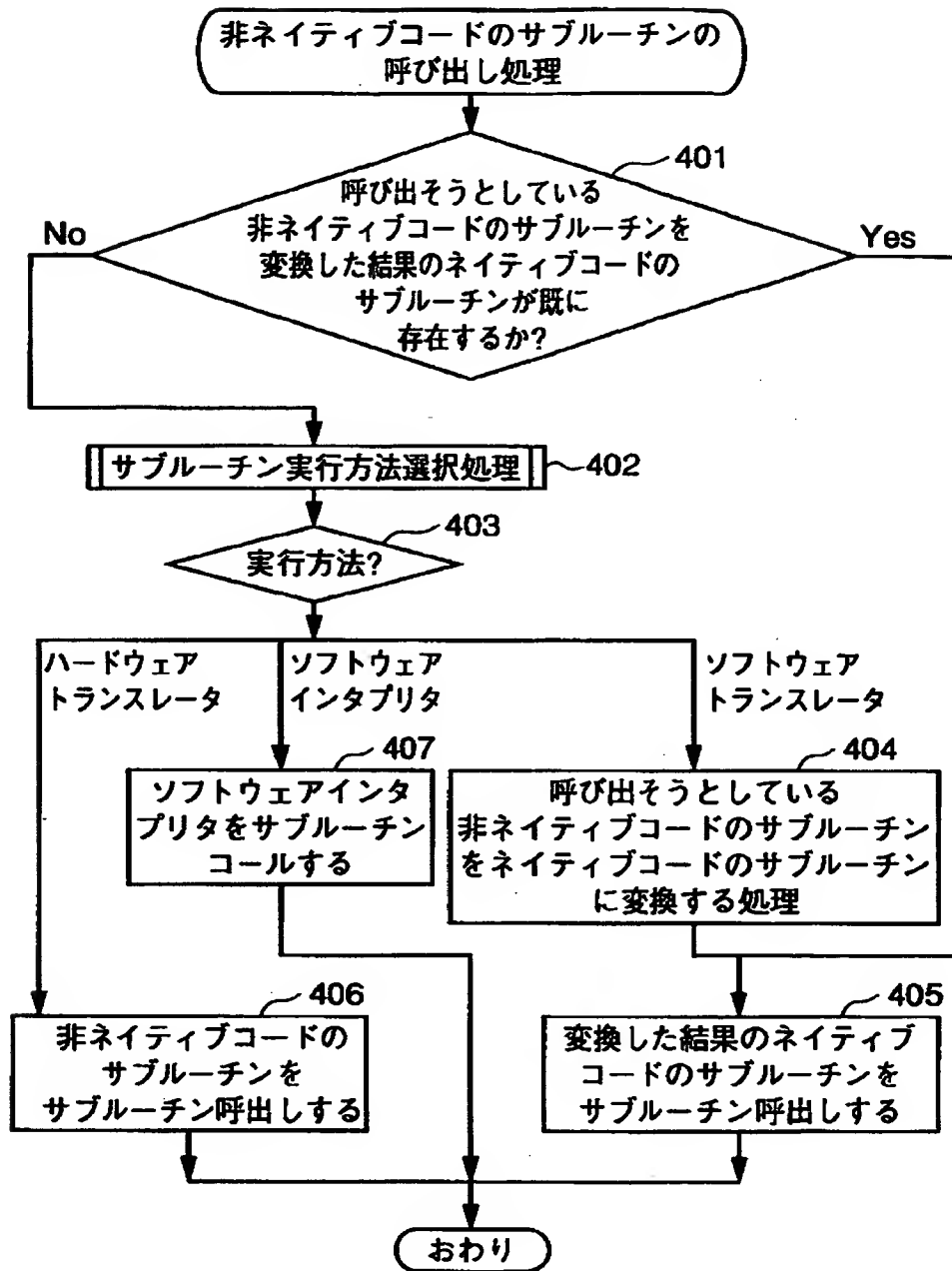
【図 6】

番地	Java/バイトコード	意味
0	iload_0	ローカル変数0をスタックにプッシュする
1	iload_1	ローカル変数1をスタックにプッシュする
2	iadd	スタックトップのふたつの整数をポップし、加算結果をスタックにプッシュする
3	istore_2	スタックトップの整数をポップし、ローカル変数2にストアする
4	iconst_1	1をスタックにプッシュする
5	iload_0	ローカル変数0をスタックにプッシュする
6	ifge 21	スタックトップをポップし、その値が0以上ならば21番地へジャンプする
9	iconst_2	2をスタックにプッシュする
10	iload_0	ローカル変数0をスタックにプッシュする
11	iload_1	ローカル変数1をスタックにプッシュする
12	iconst_3	3をスタックにプッシュする
13	iload_2	ローカル変数2をスタックにプッシュする
14	iadd	スタックトップのふたつの整数をポップし、それらを加算した結果をスタックにプッシュする
15	ldiv	スタックトップのふたつの整数をポップし、スタックトップにあった方の整数でもう一方の整数を割った結果をスタックにプッシュする
16	iadd	スタックトップのふたつの整数をポップし、それらを加算した結果をスタックにプッシュする
17	imul	スタックトップのふたつの整数をポップし、それらを乗算した結果をスタックにプッシュする
18	goto 28	28番地へジャンプする
21	iload_0	ローカル変数0をスタックにプッシュする
22	iconst_1	1をスタックにプッシュする
23	isub	スタックトップのふたつの整数をポップし、減算結果をスタックにプッシュする
24	iload_2	ローカル変数2をスタックにプッシュする
25	invokestatic <int F(int, int)>	メソッド int F(int, int) を呼び出す
28	iadd	スタックトップのふたつの整数をポップし、加算結果をスタックにプッシュする
29	ireturn	スタックトップをポップし、その値をリターン値としてサブルーチンの呼び出し元へジャンプする

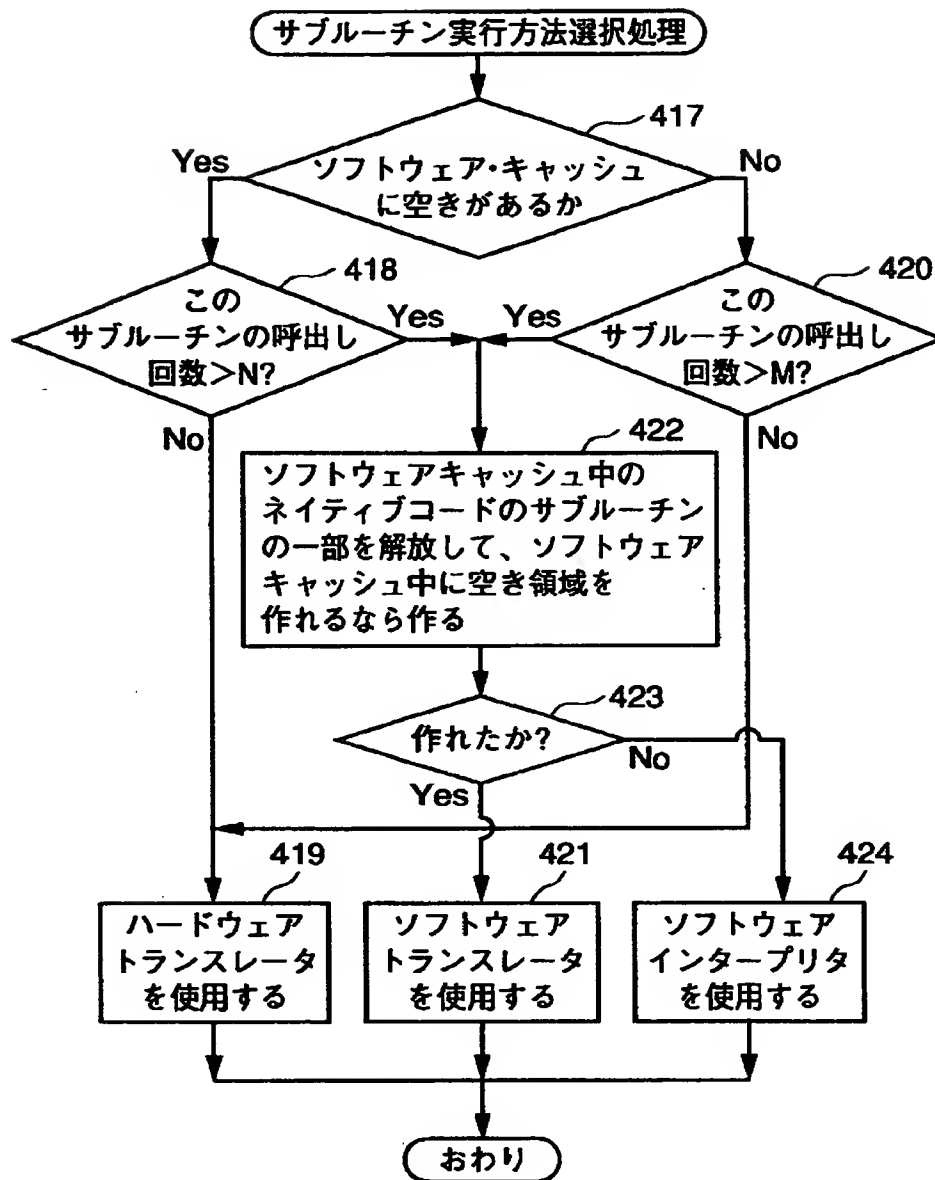
【図 7】



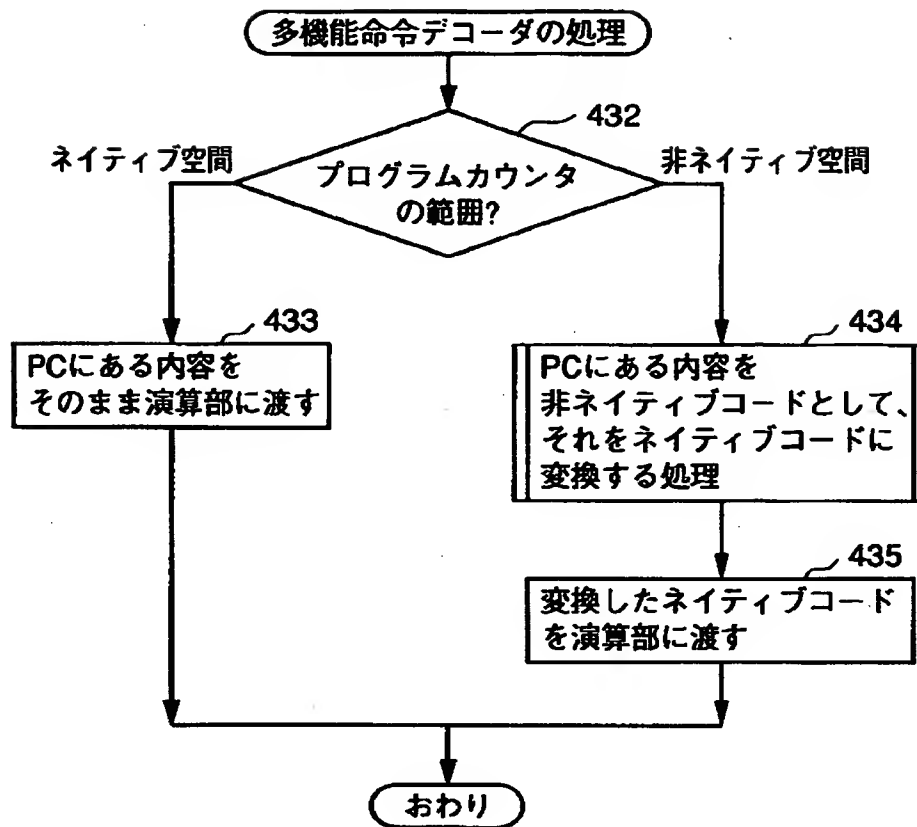
【図 8】



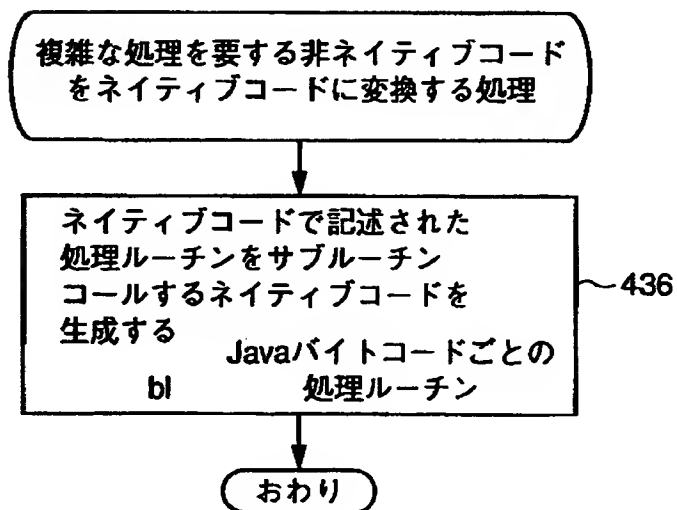
【図 9】



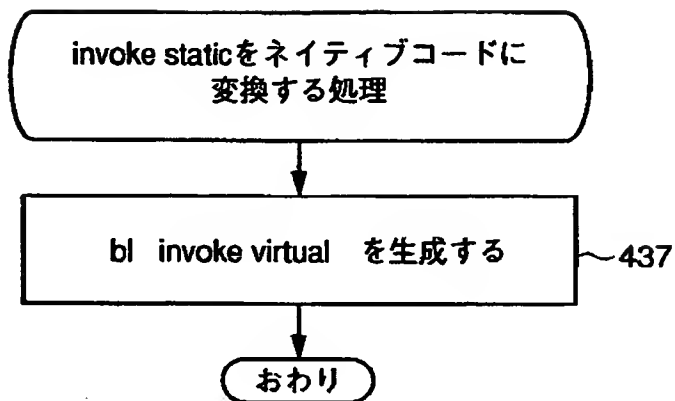
【図 1 0】



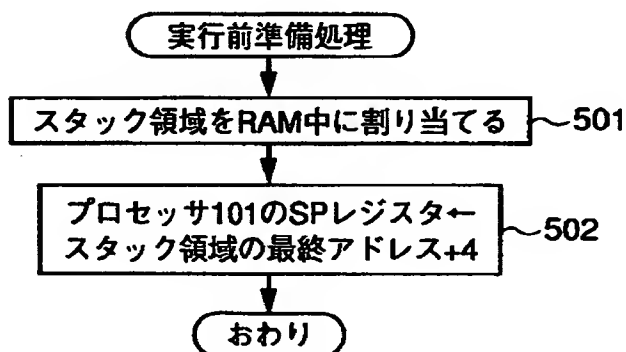
【図 1 1】



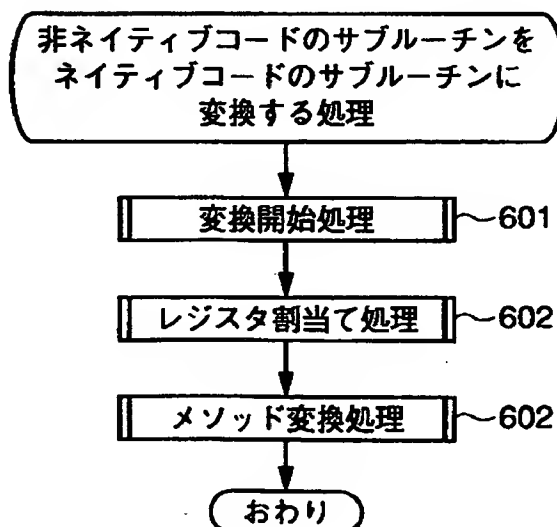
【図12】



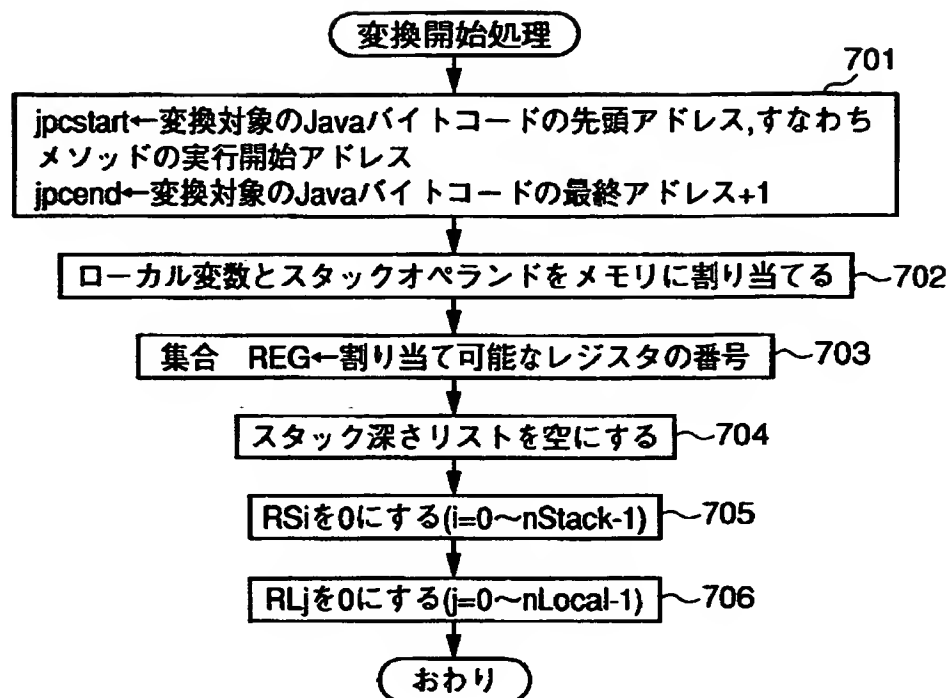
【図13】



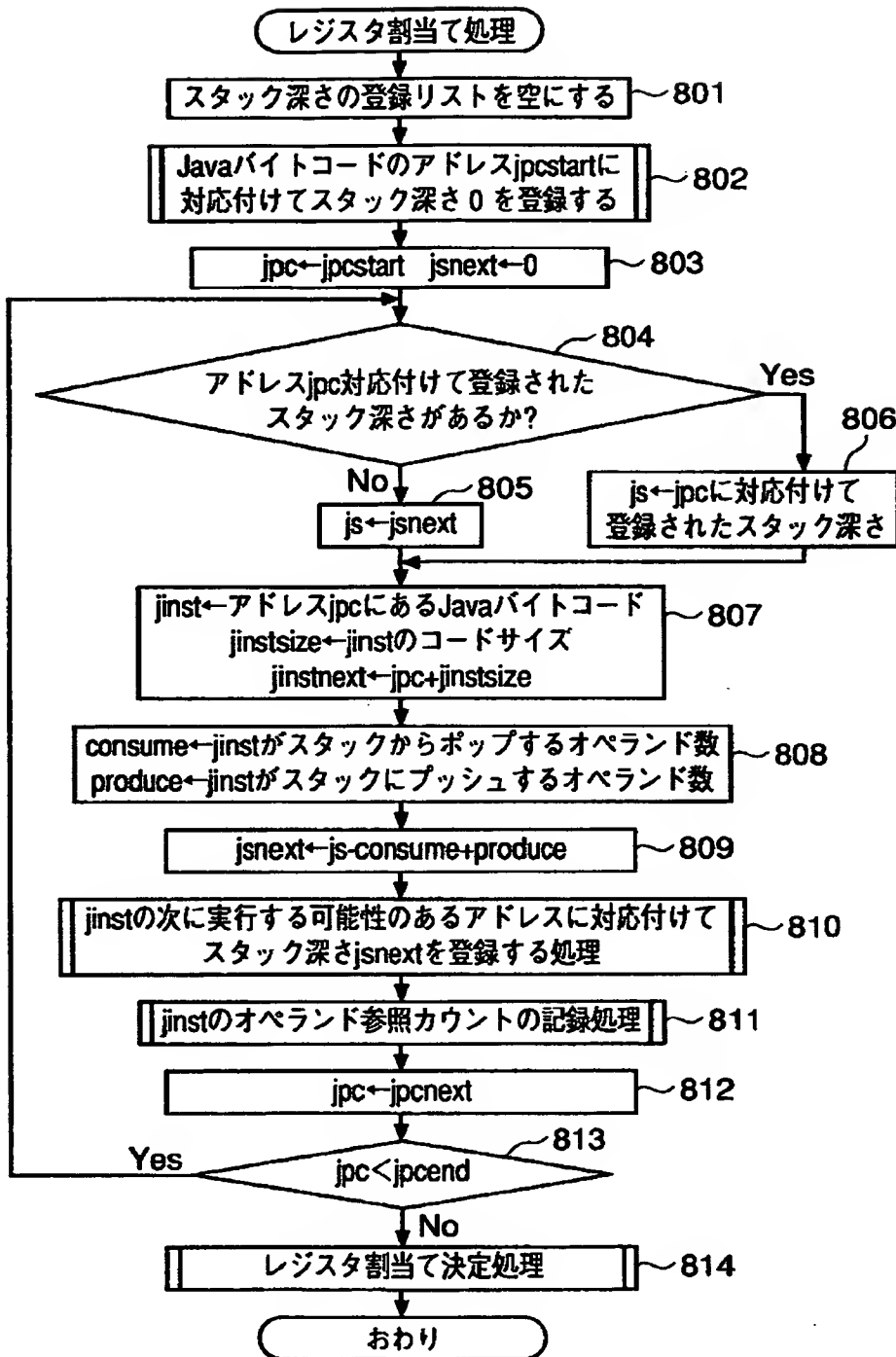
【図14】



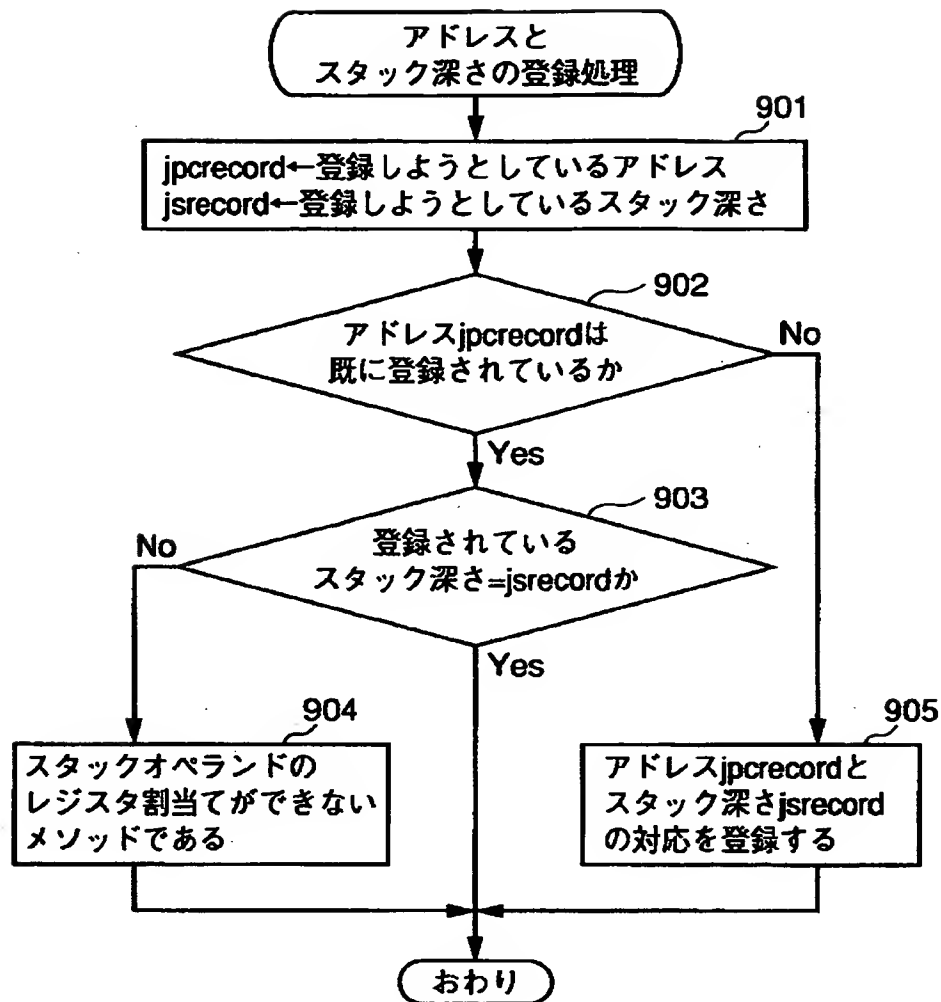
【図 1 5】



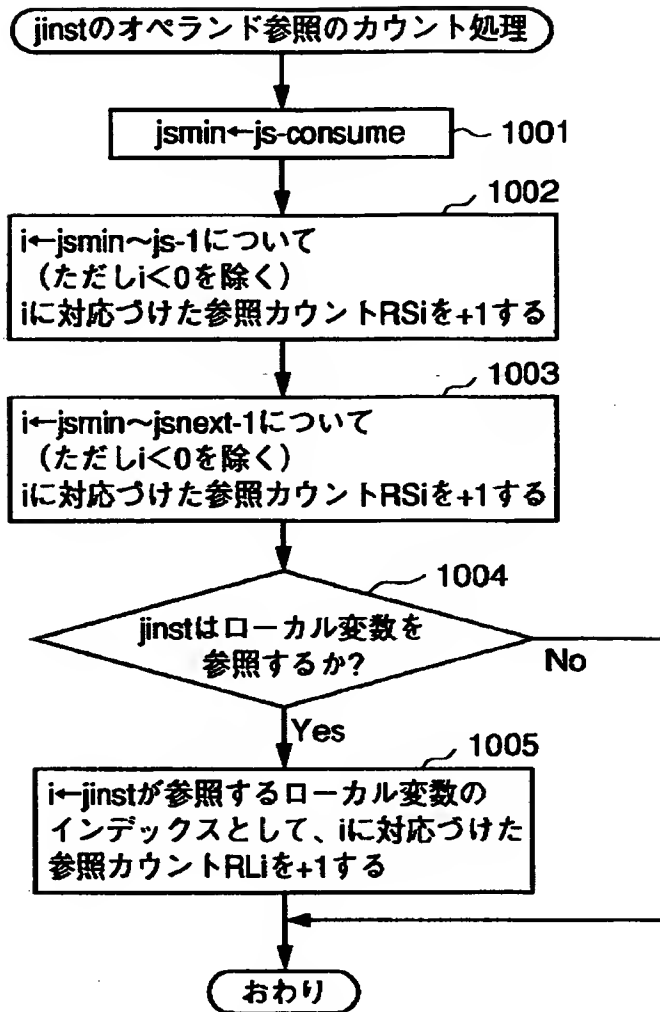
【図 1 6】



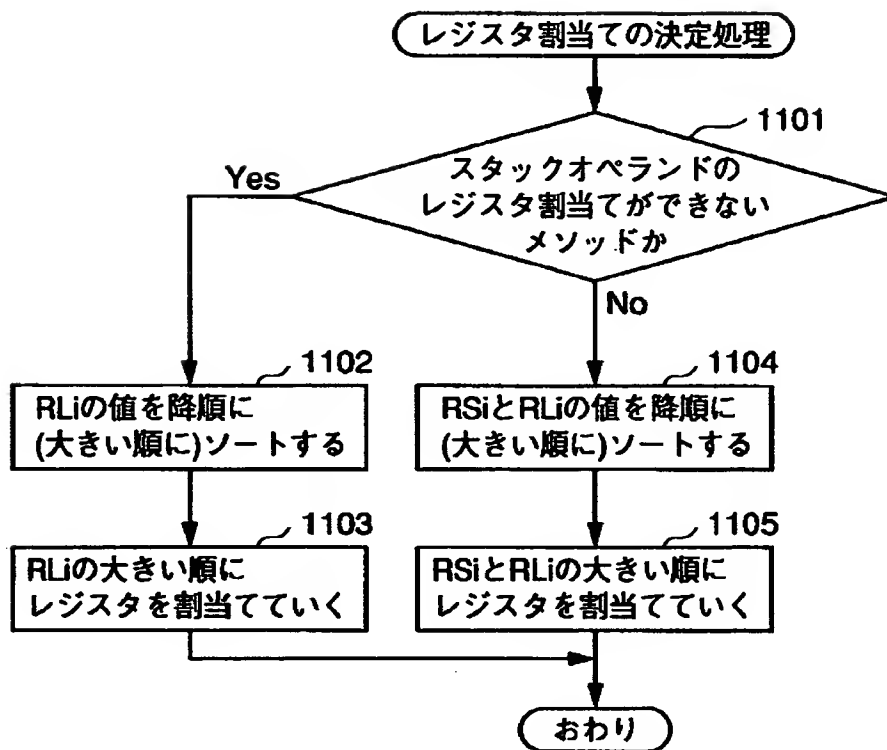
【図 1 7】



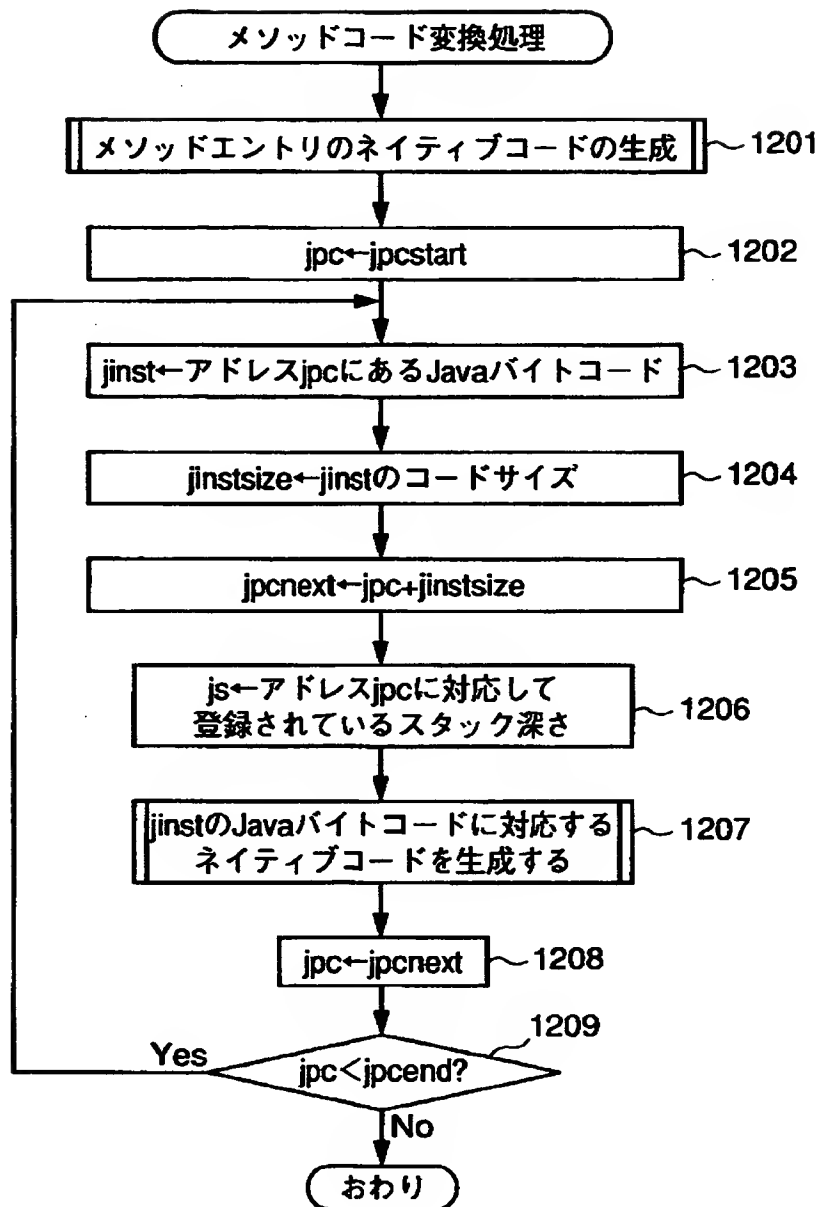
【図 18】



【図19】



【図 2 0】



【図 2 1】

Javaバイトコード	オペランド割り当て	ネイティブコード
	S<js>	
iconst_<n>	レジスタ	ldi S<js>, #n
	メモリ	ldi r0, #n
		st r0, @S<js>

【図 2 2】

Javaバイトコード	オペランド割り当て		ネイティブコード
	S<js>	L<n>	
iload_<n>	レジスタ	レジスタ	mv S<js>, L<n>
	レジスタ	メモリ	ld S<js>, @L<n>
	メモリ	レジスタ	st L<n>, @S<js>
	メモリ	メモリ	ld r0, @L<n> st r0, @S<js>

【図 2 3】

Javaバイトコード	オペランド割り当て		ネイティブコード
	S<js-1>	L<n>	
istore_<n>	レジスタ	レジスタ	mv L<n>, S<js-1>
	レジスタ	メモリ	st S<js-1>, @L<n>
	メモリ	レジスタ	ld L<n>, @S<js-1>
	メモリ	メモリ	ld r0, @S<js-1> st r0, @L<n>

【図 2 4】

Javaバイトコード	オペランド割り当て		ネイティブコード
	S<js-2>	S<js-1>	
iadd	レジスタ	レジスタ	add S<js-2>, S<js-1>
	レジスタ	メモリ	ld r0, @S<js-1> add S<js-2>, r0
	メモリ	レジスタ	ld r0, @S<js-2> add r0, S<js-1> st r0, @S<js-2>
	メモリ	メモリ	ld r0, @S<js-2> ld r1, @S<js-1> add r0, r1 st r0, @S<js-2>

【図 2 5】

Javaバイトコード	オペランド割り当て		ネイティブコード
	S<js-1>		
ifge X	レジスタ		bgez S<js-1>, TX
	メモリ		ld r0, @S<js-1> bgez r0, TX

TXはX番地のJavaバイトコードに対して生成されたネイティブコードのアドレスである。

【図 2 6】

Java/バイトコード	-	ネイティブコード
goto X	-	bra TX

TXはX番地のJava/バイトコードに対して生成されたネイティブコードのアドレスである。

【図 2 7】

Java/バイトコード	オペランド割り当て	ネイティブコード
	S<js-1>	
ireturn	レジスタ	mv r0, S<js-1> エピローグのコード
	メモリ	ld r0, @S<js-1> エピローグのコード

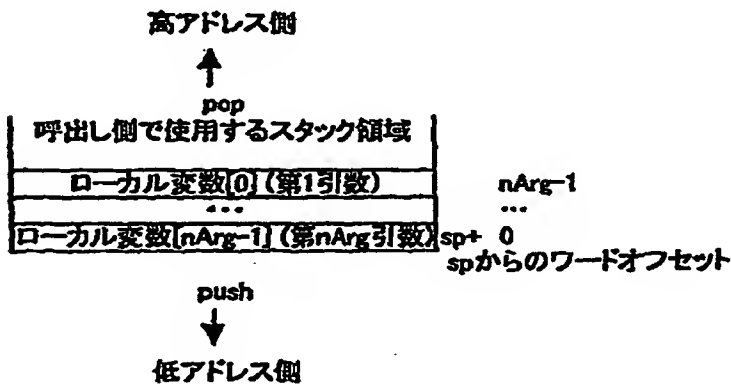
【図 2 8】

Java/バイトコード	オペランド割り当て		ネイティブコード
	S<js-2>	S<js-1>	
intokestatic <int F(int, int)>	レジスタ	レジスタ	push S<js-2> push S<js-1> ld24 r0, #method_id bl call_java_method addi sp, #8 mv S<js-1>, r0
		メモリ	push S<js-2> ld r0, @S<js-1> push r0 ld24 r0, #method_id bl call_java_method addi sp, #8 mv S<js-1>, r0
	メモリ	レジスタ	ld r0, @S<js-2> push r0 push S<js-1> ld24 r0, #method_id bl call_java_method addi sp, #8 st r0, @S<js-1>
	メモリ	メモリ	ld r0, @S<js-2> push r0 ld r0, @S<js-1> push r0 ld24 r0, #method_id bl call_java_method addi sp, #8 st r0, @S<js-1>

【図 2 9】

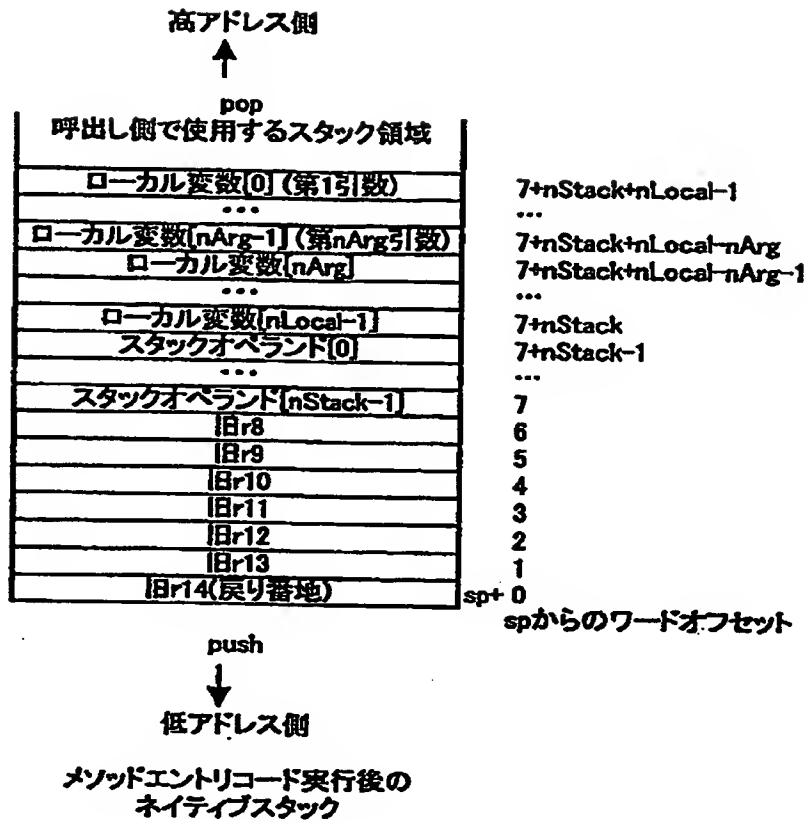
レジスタ	使用方法、使用範囲
r0-r3	計算のためにテンポラリに使用してよい。 r0と1はサブルーチンから戻る際に戻り値を格納するためにも使用する。 これらのレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r4-r7	計算のためにテンポラリに使用してよい。 これらのレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r8-r13	オペランドスタックやローカル変数用に割り当てて使用する。 これらのレジスタの値はサブルーチンの呼出し前後で保存する。
r14 (lr)	リンクレジスタ。 サブルーチン呼出しの際に、戻り番地を格納するために使用する。 このレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r15 (sp)	スタックポインタ。 このレジスタの値はサブルーチンの呼出し前後で保存する。

【図 3 0】

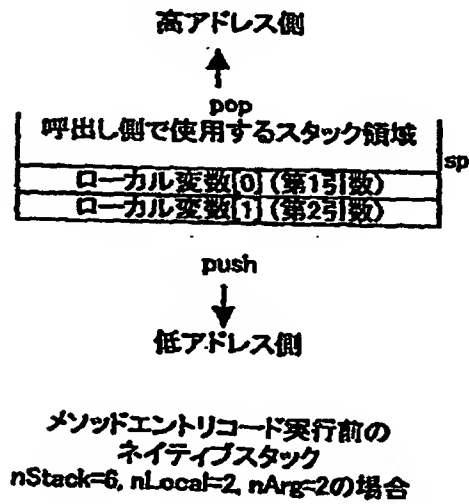


メソッドエントリコード実行前の
ネイティブスタック

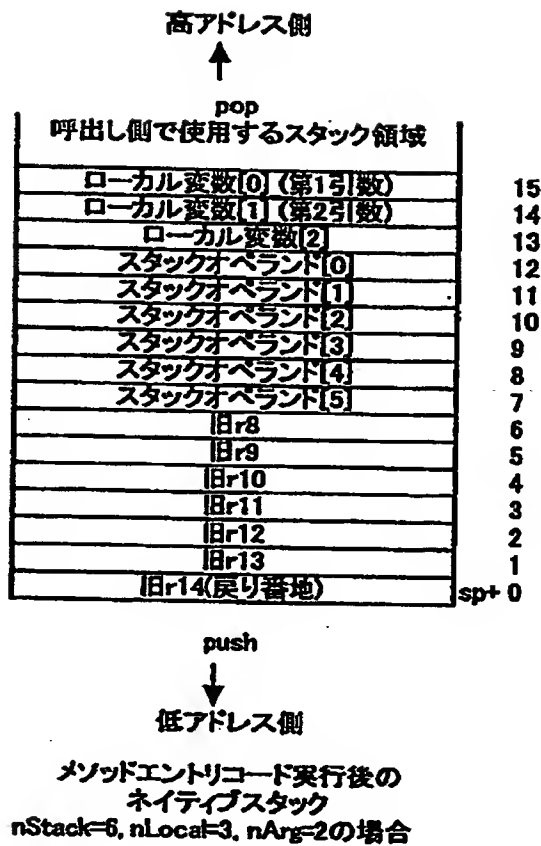
【図 3 1】



【図 3 2】



【図 3 3】



【図 3 4】

記号	オペランド	レジスタ割当て	
		変換開始処理直後	レジスタ割当て処理後
L<0>	ローカル変数[0] (第1引数)	(32 , SP)	R 13
L<1>	ローカル変数[1] (第2引数)	(28 , SP)	(56 , SP)
L<2>	ローカル変数[2]	(24 , SP)	(52 , SP)
S<0>	スタックオペランド[0]	(20 , SP)	R 9
S<1>	スタックオペランド[1]	(16 , SP)	R 8
S<2>	スタックオペランド[2]	(12 , SP)	R 10
S<3>	スタックオペランド[3]	(8 , SP)	R 11
S<4>	スタックオペランド[4]	(4 , SP)	R 12
S<5>	スタックオペランド[5]	(0 , SP)	(28 , SP)

【図 35】

状態	jpc js	jinst	jinstsize	jpcnext	consume	produce	jsnext	スタック深さリストに登録される命令アドレスとスタック深さ	RS										RL	
									0	1	2	3	4	5	0	1	2			
(1)	0							[0,0]	0	0	0	0	0	0	0	0	0	0	0	
(2)	0	0 iload 0	1	1	0	1	1	[1,1]		1	0	0	0	0	0	1	0	0		
(3)	1	1 iload 1	1	2	0	1	2	[2,2]		1	1	0	0	0	0	1	1	0		
(4)	2	2 iadd	1	3	2	1	1	[3,1]		3	2	0	0	0	0	1	1	0		
(5)	3	1 istore 2	1	4	1	0	0	[4,0]		4	2	0	0	0	0	1	1	0		
(6)	4	0 iconst 1	1	5	0	1	1	[5,1]		5	2	0	0	0	0	1	1	1		
(7)	5	1 iload 0	1	6	0	1	2	[6,2]		5	3	0	0	0	0	2	1	1		
(8)	6	2 ifge 21	3	9	1	0	1	[9,1][21,1]		5	4	0	0	0	0	2	1	1		
(9)	9	1 iconst 2	1	10	0	1	2	[10,2]		5	5	0	0	0	0	2	1	1		
(10)	10	2 iload 0	1	11	0	1	3	[11,3]		5	5	1	0	0	0	3	1	1		
(11)	11	3 iload 1	1	12	0	1	4	[12,4]		5	5	1	1	0	0	3	2	1		
(12)	12	4 iconst 3	1	13	0	1	5	[13,5]		5	5	1	1	1	0	3	2	1		
(13)	13	5 iload 2	1	14	0	1	6	[14,6]		5	5	1	1	1	1	3	2	2		
(14)	14	6 iadd	1	15	2	1	5	[15,5]		5	5	1	1	3	2	3	2	2		
(15)	15	3 idiv	1	16	2	1	4	[16,4]		5	5	1	3	4	2	3	2	2		
(16)	16	4 iadd	1	17	2	1	3	[17,3]		5	5	3	4	4	2	3	2	2		
(17)	17	3 imul	1	18	2	1	2	[18,2]		5	7	4	4	4	2	3	2	2		
(18)	18	2 goto 28	3	21	0	0	2	[28,2]		5	7	4	4	4	2	3	2	2		

【 36 】

(19)	21	1	load 0	1	22	0	1	2	[22,2]	5	8	4	4	2	4	2	2
(20)	22	2	lconst 1	1	23	0	1	3	[23,3]	5	8	5	4	2	4	2	2
(21)	23	3	lsub	1	24	2	1	2	[24,2]	5	10	6	4	2	4	2	2
(22)	24	2	load 2	1	25	0	1	3	[25,3]	5	10	7	4	2	4	2	3
(23)	25	3	invoke stat1 c	3	28	2	1	2	[28,2]	5	12	8	4	2	4	2	3
			<int F(int, 2 ladd	1	29	2	1	1	[29,1]	7	13	8	4	2	4	2	3
(24)	28	2	ladd	1	29	2	1	1	[29,1]	7	13	8	4	2	4	2	3
(25)	29	1	lreturn	1	30	1	0	0		8	13	8	4	2	4	2	3

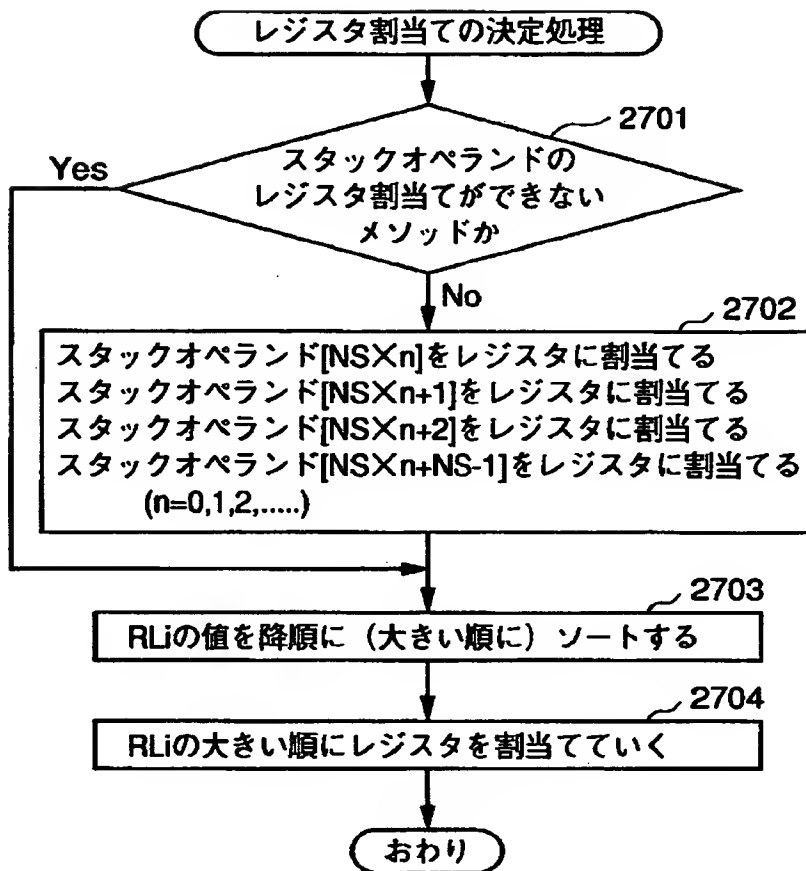
【図 37】

状態	jpc	jinst	jinstsize	jpcnext	js	オペランド
(1)						addi sp, #-(nlocal-nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld l<0>, 0((nlocal+nStack+nSave-1)*4,sp)
(2)	0	iload_0	1	1	0	mv s<0>, l<0>
(3)	1	iload_1	1	2	1	ld s<1>, 0l<1>
(4)	2	ladd	1	3	2	add s<0>, s<1>
(5)	3	istore_2	1	4	1	st s<0>, 0l<2>
(6)	4	lconst_1	1	5	0	ldi s<0>, #1
(7)	5	iload_0	1	6	1	mv s<1>, l<0>
(8)	6	lfige_21	3	9	2	bgez s<1>, T21
(9)	9	lconst_2	1	10	1	ldi s<1>, #2
(10)	10	iload_0	1	11	2	mv s<2>, l<0>
(11)	11	iload_1	1	12	3	ld s<3>, 0l<1>
(12)	12	lconst_3	1	13	4	ldi s<4>, #3
(13)	13	iload_2	1	14	5	ld r0, 0l<2> st r0, 0s<5>
(14)	14	ladd	1	15	6	ld r0, 0s<5> add s<4>, r0
(15)	15	ldiv	1	16	5	div s<3>, s<4>
(16)	16	ladd	1	17	4	add s<2>, s<3>
(17)	17	lmul	1	18	3	mul s<1>, s<2>
(18)	18	goto_28	3	21	2	bra T28
(19)	21	iload_0	1	22	1	T21: mv s<1>, l<0>
(20)	22	lconst_1	1	23	2	ldi s<2>, #1
(21)	23	lsub	1	24	3	sub s<1>, s<2>

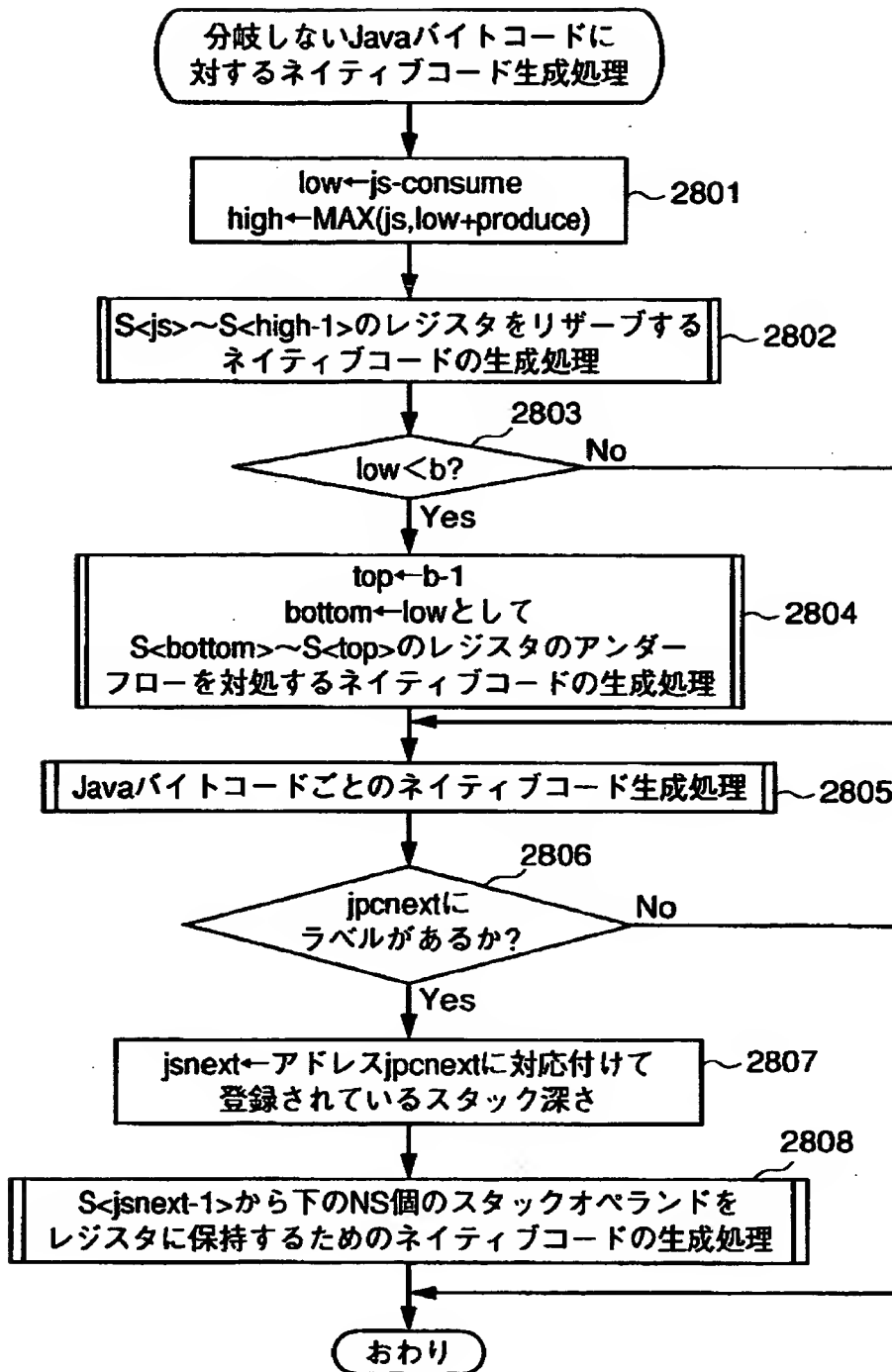
【図 38】

(22)	24	lload 2	1	25	2	ld s<2>, @L<2>
(23)	25	invokestatic <int F(int, int)>	3	28	3	push s<2> push s<1> ld24 r0, #methodId j1 callJavaMethod addi sp, #8 mv s<1>, r0
(24)	28	ladd	1	29	2	T28: add s<0>, s<1>
(25)	29	lreturn	1	30	1	mv r0, s<0> pop lr pop r13 pop r12 pop r11 pop r10 pop r9 pop r8 addi sp, #(nlocal-nArg+nStack)*4 jmp lr

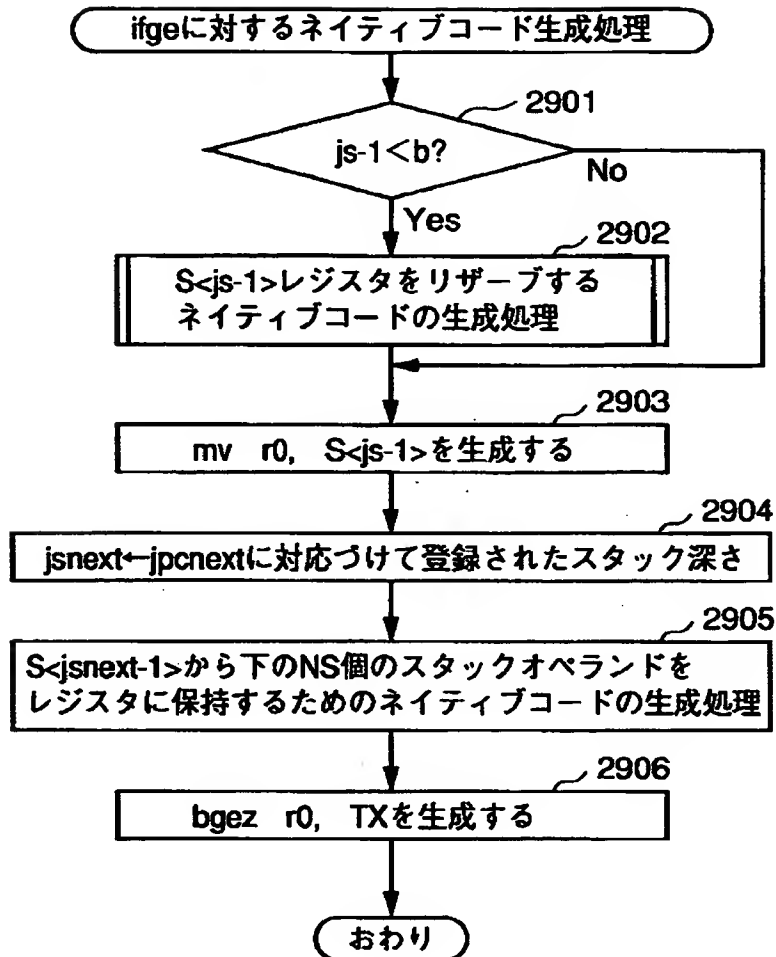
【図 3 9】



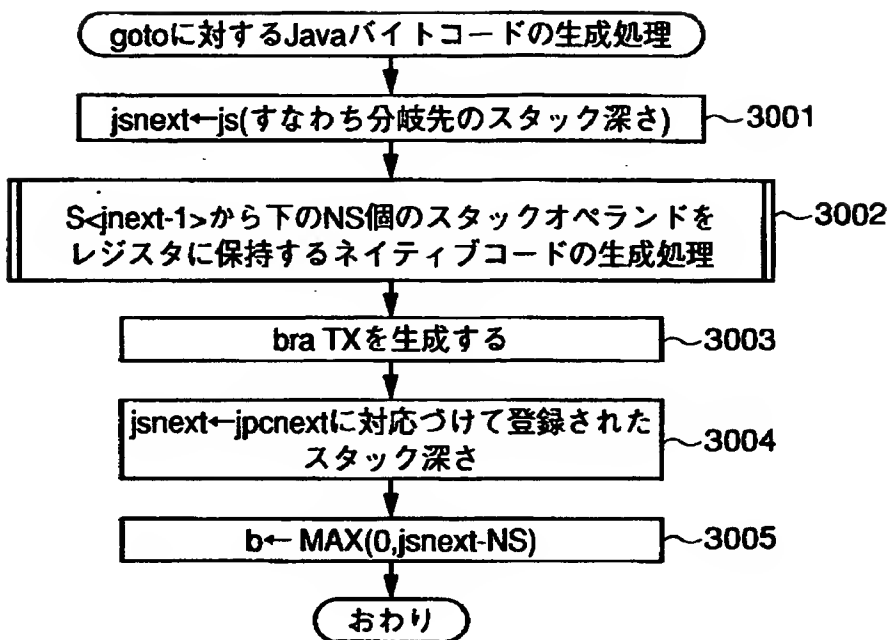
【図 40】



【図 4 1】



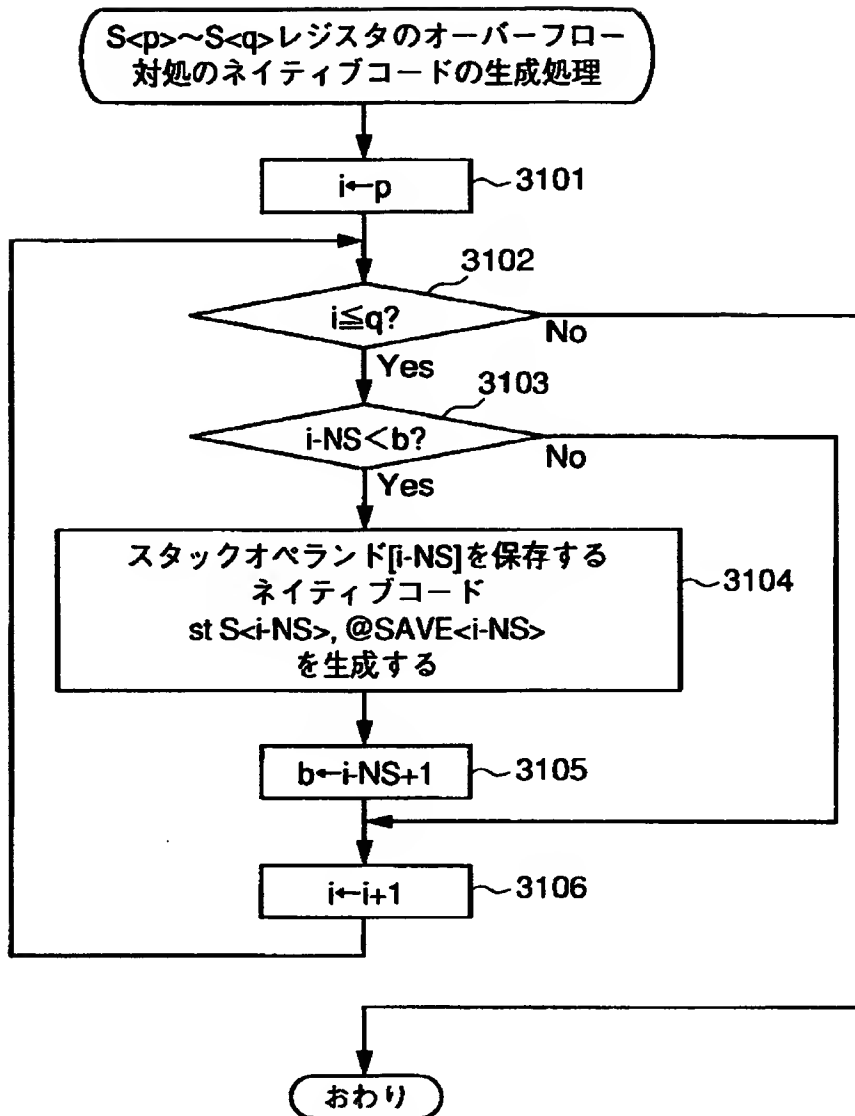
【図 4 2】



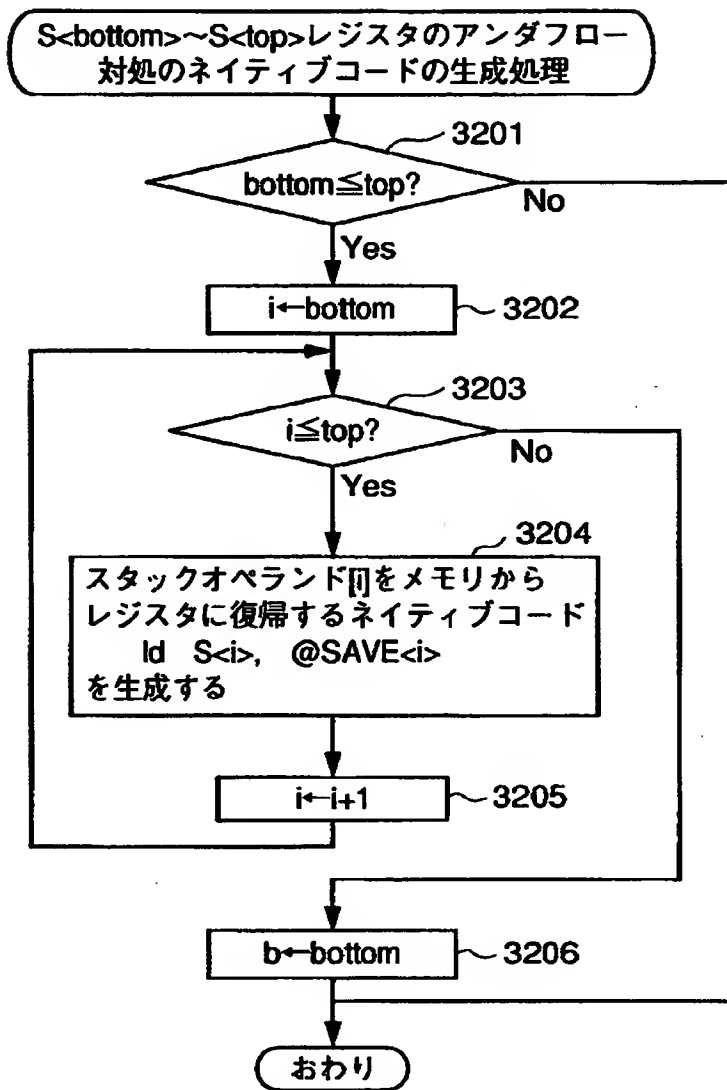
【図 4 3】

レジスタ	使用方法、使用範囲
r0-r3	計算のためにテンポラリに使用してよい。 r0とr1はサブルーチンから戻る際に戻り値を格納するためにも使用する。 これらのレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r4-r7	計算のためにテンポラリに使用してよい。 これらのレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r8-r13	オペランドスタック用に割り当てて使用する。 r8にはオペランドスタック[4n]を割り当てる。 r9にはオペランドスタック[4n+1]を割り当てる。 r10にはオペランドスタック[4n+2]を割り当てる。 r11にはオペランドスタック[4n+3]を割り当てる。(n=0,1,2,...) これらのレジスタの値はサブルーチンの呼出し前後で保存する。
r12-r13	ローカル変数用に割り当てて使用する。 これらのレジスタの値はサブルーチンの呼出し前後で保存する。
r14 (lr)	リンクレジスタ。 サブルーチン呼出しの際に、戻り番地を格納するために使用する。 このレジスタの値はサブルーチンの呼出し前後で保存しなくてよい。
r15 (sp)	スタックポインタ。 このレジスタの値はサブルーチンの呼出し前後で保存する。

【図 4 4】



【図 45】



【図 4 6】

記号	オペランド	レジスタ割当て	
		変換開始処理 直後	レジスタ割当て処理 後
L<0>	ローカル変数[0] (第1引数)	(60 ,SP)	R 12
L<1>	ローカル変数[1] (第2引数)	(56 ,SP)	(56 ,SP)
L<2>	ローカル変数[2]	(52 ,SP)	R 13
S<0>	スタックオペランド[0]	(48 ,SP)	R 8
S<1>	スタックオペランド[1]	(44 ,SP)	R 9
S<2>	スタックオペランド[2]	(40 ,SP)	R 10
S<3>	スタックオペランド[3]	(36 ,SP)	R 11
S<4>	スタックオペランド[4]	(32 ,SP)	R 8
S<5>	スタックオペランド[5]	(28 ,SP)	R 9

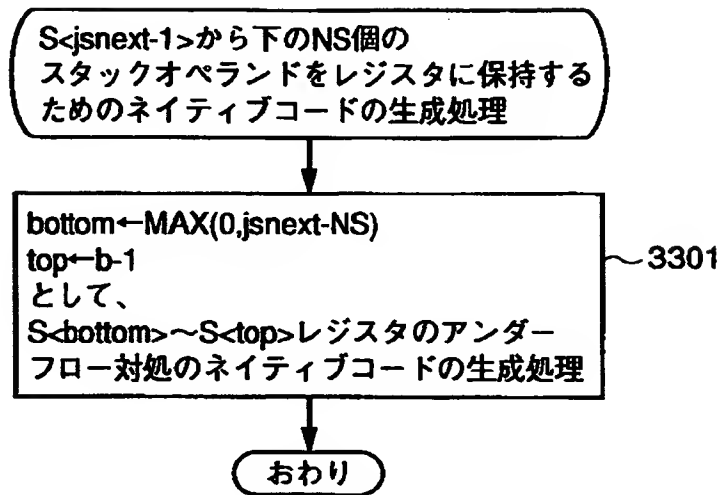
【図 47】

状態 jpc	jinst	jinstsize	jpcnext	js	b	consume	produce	low	high	ネイティブコード
(1)										addl sp, 4(nlocal-nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld L<0>, 0((nlocal+nStack+nSave-1)*4,sp) ld L<1>, 0((nlocal+nStack+nSave-2)*4,sp) mv S<0>, L<0>
(2)	0 iload 0	1	1	0	0	0	1	0	1	
(3)	1 iload 1	1	2	1	0	0	1	1	2	
(4)	2 iadd	1	3	2	0	2	1	0	2	
(5)	3 istore 2	1	4	1	0	1	0	0	1	
(6)	4 iconst 1	1	5	0	0	0	1	0	1	
(7)	5 iload 0	1	6	1	0	0	1	1	2	
(8)	6 ifge 21	3	9	2	0	1	0	1	2	
(9)	9 iconst 2	1	10	1	0	0	1	1	2	
(10)	10 iload 0	1	11	2	0	0	1	2	3	
(11)	11 iload 1	1	12	3	0	0	1	3	4	
(12)	12 iconst 3	1	13	4	1	0	1	4	5	
(13)	13 iload 2	1	14	5	2	0	1	5	6	
(14)	14 iadd	1	15	6	2	2	1	4	6	
(15)	15 idiv	1	16	5	2	2	1	3	5	
(16)	16 iadd	1	17	4	2	2	1	2	4	
(17)	17 imul	1	18	3	1	2	1	1	3	
(18)	18 goto 28	3	21	2	0	0	0	2	2	
(19)	21 iload 0	1	22	1	0	0	1	1	2	
(20)	22 iconst 1	1	23	2	0	0	1	2	3	
(21)	23 isub	1	24	3	0	2	1	1	3	
(22)	24 iload 2	1	25	2	0	0	1	2	3	

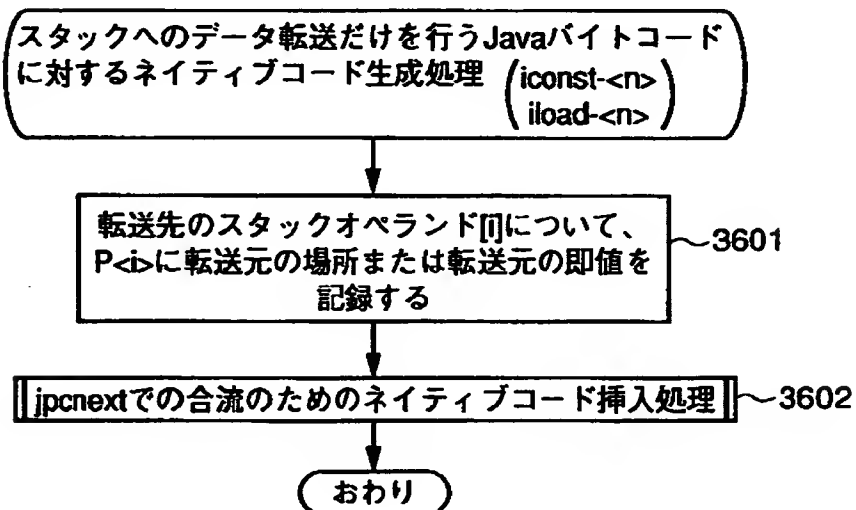
【図 48】

(23)	25	invokestatic <int F(int, int)>	3	28	3	0	2	1	1	3	push S<2> push S<1> ld24 r0, #methodId jl callJavaMethod addi sp, #8 mv S<1>, r0
(24)	28	ladd	1	29	2	0	2	1	0	2	728: add S<0>, S<1>
(25)	29	ireturn	1	30	1	0	1	0	0	1	mv r0, S<0> pop lr pop r13 pop r12 pop r11 pop r10 pop r9 pop r8 addi sp, #((nLocal-nArg+nStack)*4) jmp lr

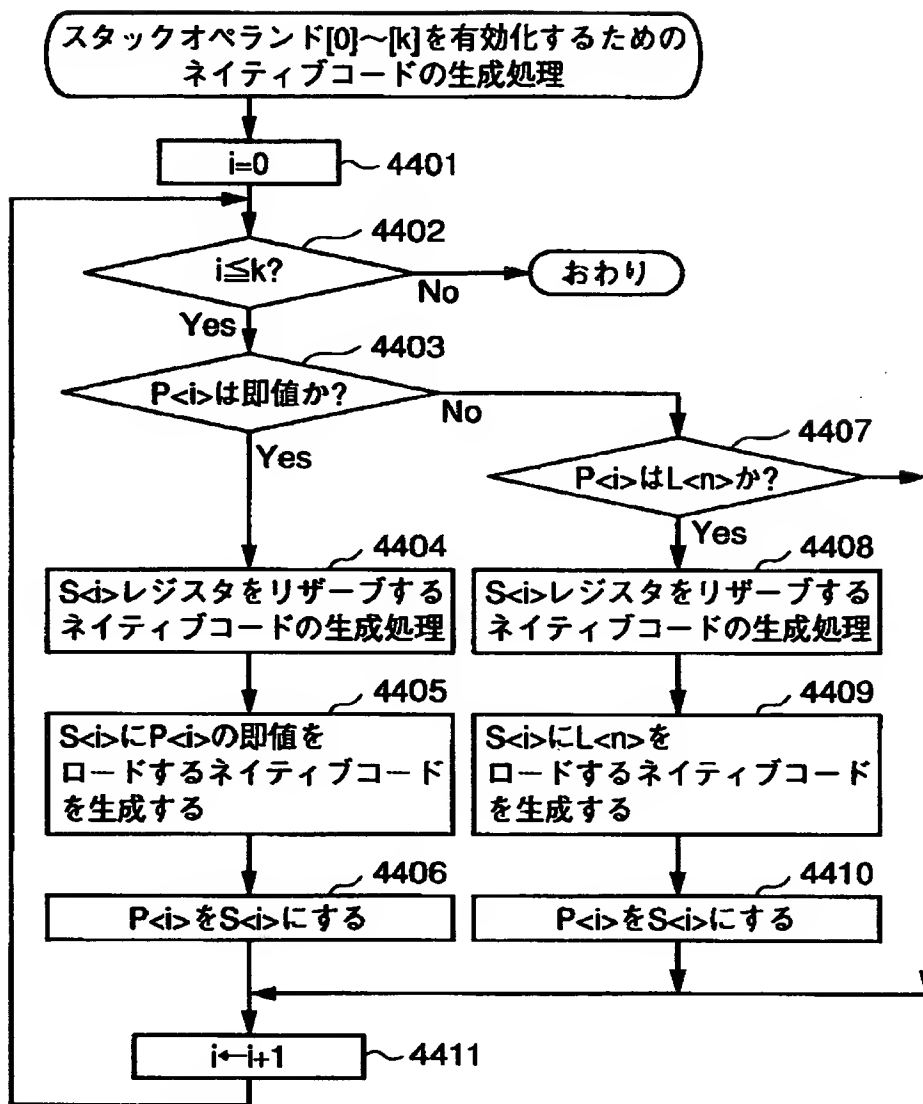
【図 4 9】



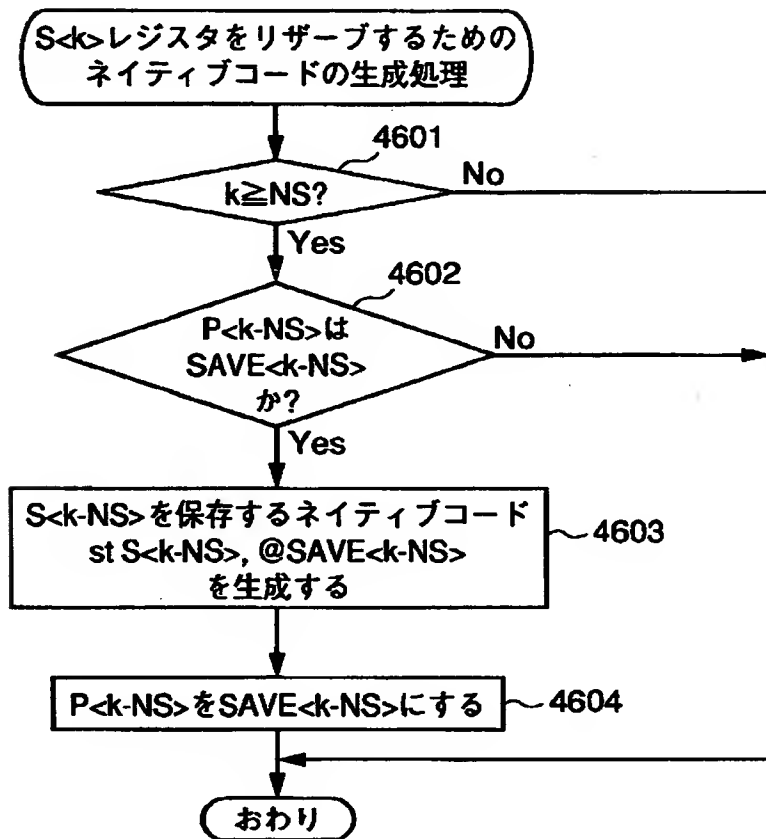
【図 5 0】



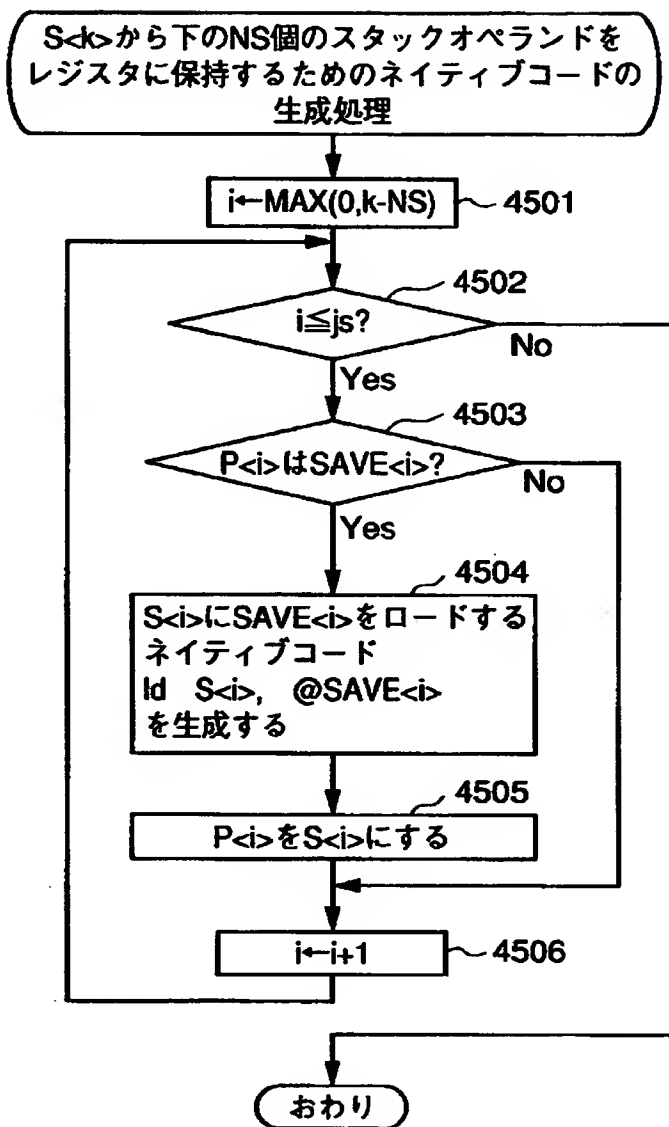
【図 5 1】



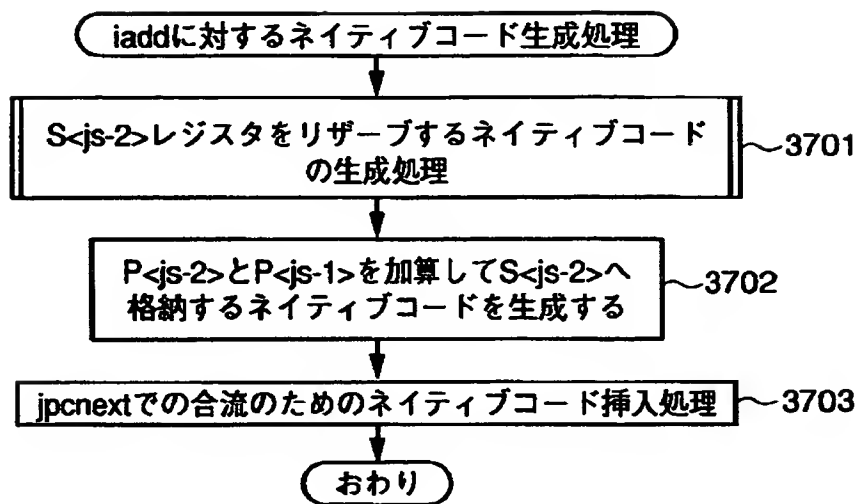
【図 5 2】



【図 53】



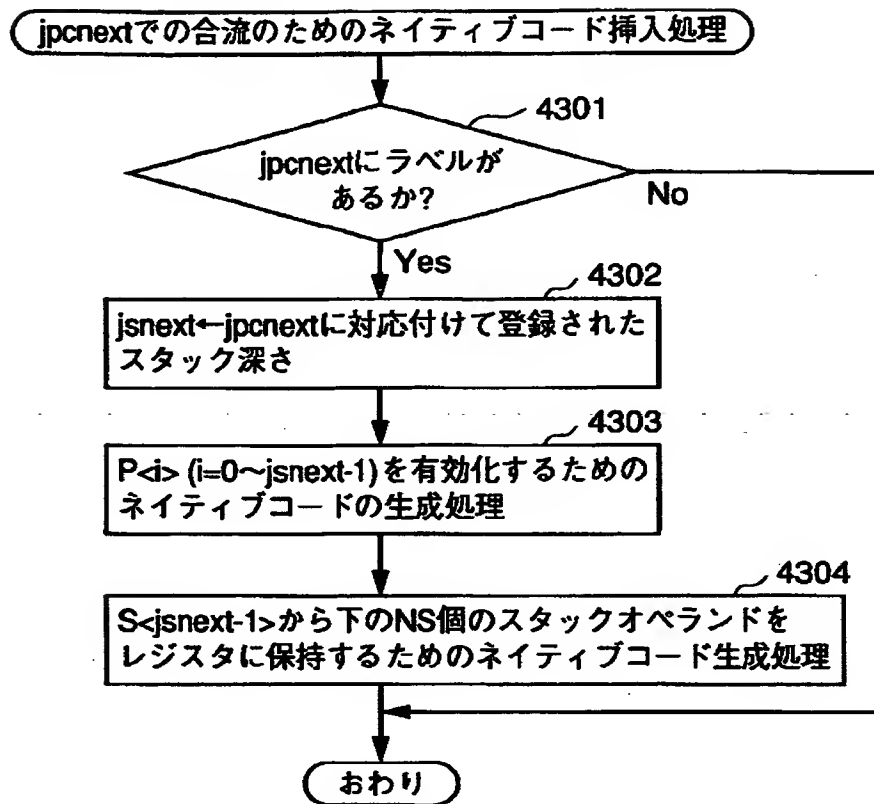
【図 5 4】



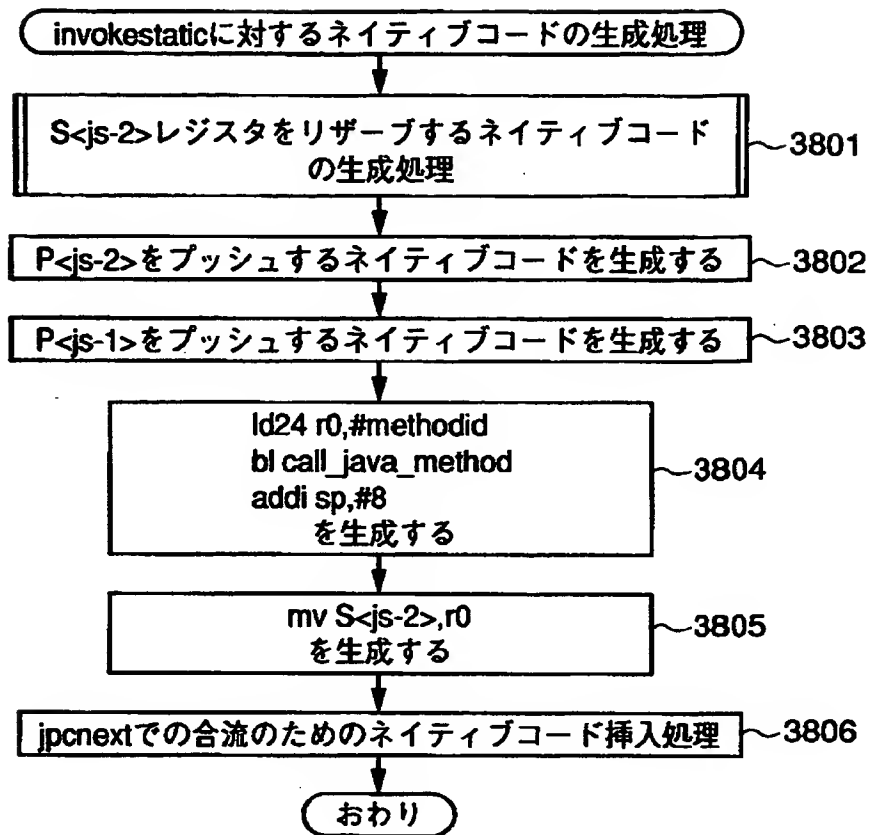
【図55】

Java バイトコード	オペランドの範囲		ケース 番号	ネイティブコード	その他の処理
iadd	P<j>a-2>	P<j>a-1>	1	なし	P<j>a-2>にP<j>a-2>+P<j>a-1>の 和値を記録する
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	2	addl S<j>a-2>, S<j>a-1>, @P<j>a-2>	
	SAVE<j>a-1>		3	ld r0, @SAVE<j>a-1>	
	ローカル変数 L<ω> (レジスタ)		4	addl S<j>a-2>, r0, @P<j>a-2>	P<j>a-2>にS<j>a-2>を記録する
	ローカル変数 L<ω> (メモリ)		5	ld S<j>a-2>, @L<ω>	
			6	addl S<j>a-2>, S<j>a-2>, @P<j>a-2>	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	7	addl S<j>a-2>, S<j>a-1>	P<j>a-2>にP<j>a-2>+P<j>a-1>の 和値を記録する
	S<j>a-1>		8	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		9	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		10	ldh S<j>a-2>, @high(P<j>a-2>)	P<j>a-2>にS<j>a-2>を記録する
	ローカル変数 L<ω> (メモリ)		11	ldh S<j>a-2>, @high(P<j>a-2>)	
			12	ldh S<j>a-2>, @high(P<j>a-2>)	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	13	ldh S<j>a-2>, @high(P<j>a-2>)	
	S<j>a-1>		14	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		15	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		16	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (メモリ)		17	ldh S<j>a-2>, @high(P<j>a-2>)	
			18	ldh S<j>a-2>, @high(P<j>a-2>)	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	19	ldh S<j>a-2>, @high(P<j>a-2>)	
	S<j>a-1>		20	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		21	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		22	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (メモリ)		23	ldh S<j>a-2>, @high(P<j>a-2>)	
			24	ldh S<j>a-2>, @high(P<j>a-2>)	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	25	ldh S<j>a-2>, @high(P<j>a-2>)	
	S<j>a-1>		26	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		27	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		28	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (メモリ)		29	ldh S<j>a-2>, @high(P<j>a-2>)	
			30	ldh S<j>a-2>, @high(P<j>a-2>)	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	31	ldh S<j>a-2>, @high(P<j>a-2>)	
	S<j>a-1>		32	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		33	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		34	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (メモリ)		35	ldh S<j>a-2>, @high(P<j>a-2>)	
			36	ldh S<j>a-2>, @high(P<j>a-2>)	
	即値(16ビット符号付範囲以下)	即値(16ビット符号付範囲以下)	37	ldh S<j>a-2>, @high(P<j>a-2>)	
	S<j>a-1>		38	ldh S<j>a-2>, @high(P<j>a-2>)	
	SAVE<j>a-1>		39	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (レジスタ)		40	ldh S<j>a-2>, @high(P<j>a-2>)	
	ローカル変数 L<ω> (メモリ)		41	ldh S<j>a-2>, @high(P<j>a-2>)	
			42	ldh S<j>a-2>, @high(P<j>a-2>)	

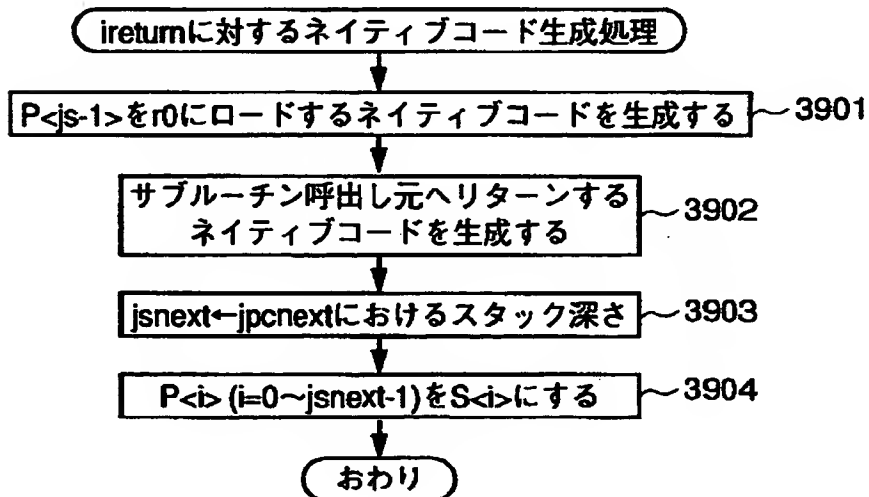
【図 5 6】



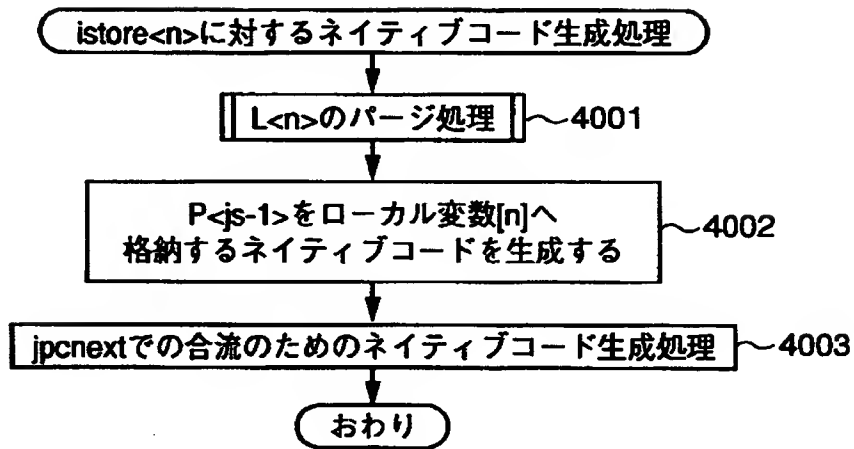
【図 5 7】



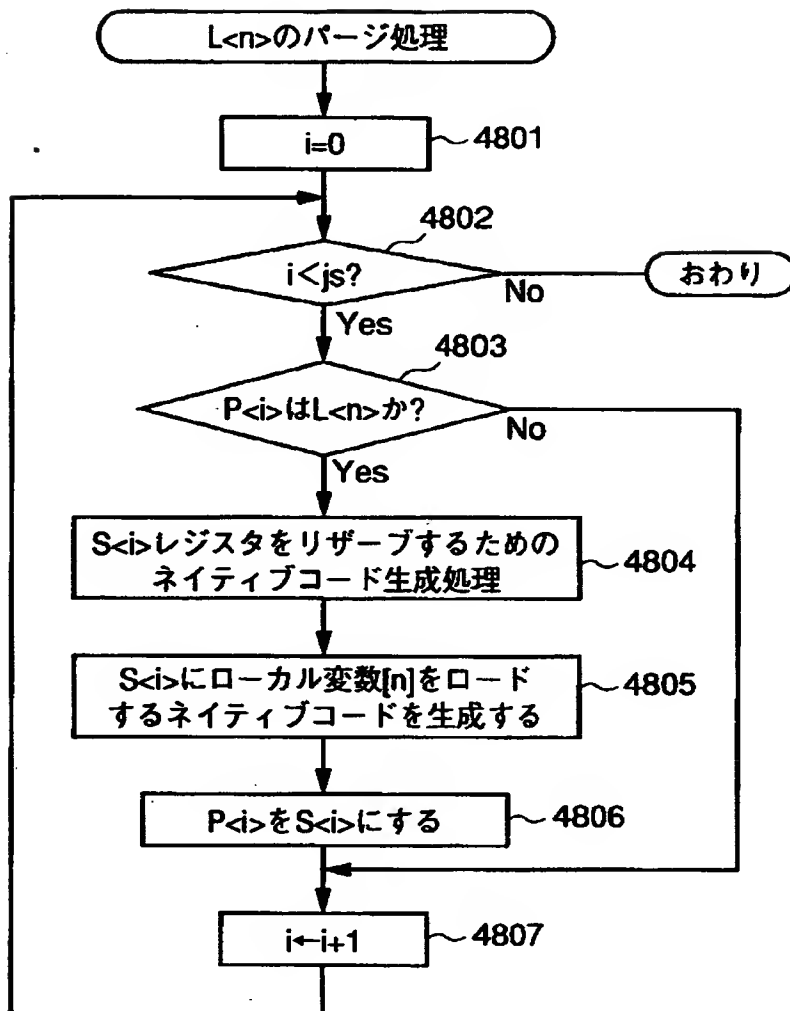
【図 5 8】



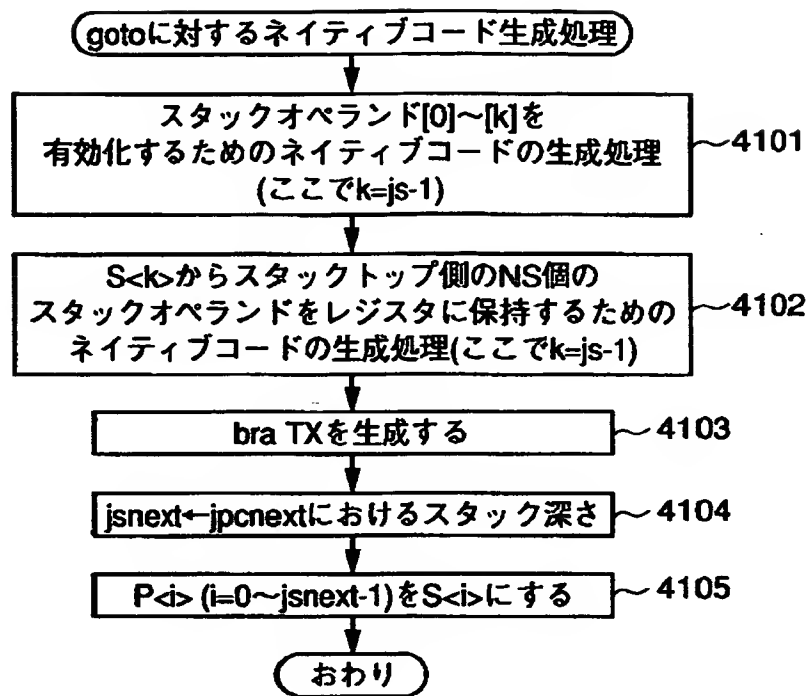
【図 5 9】



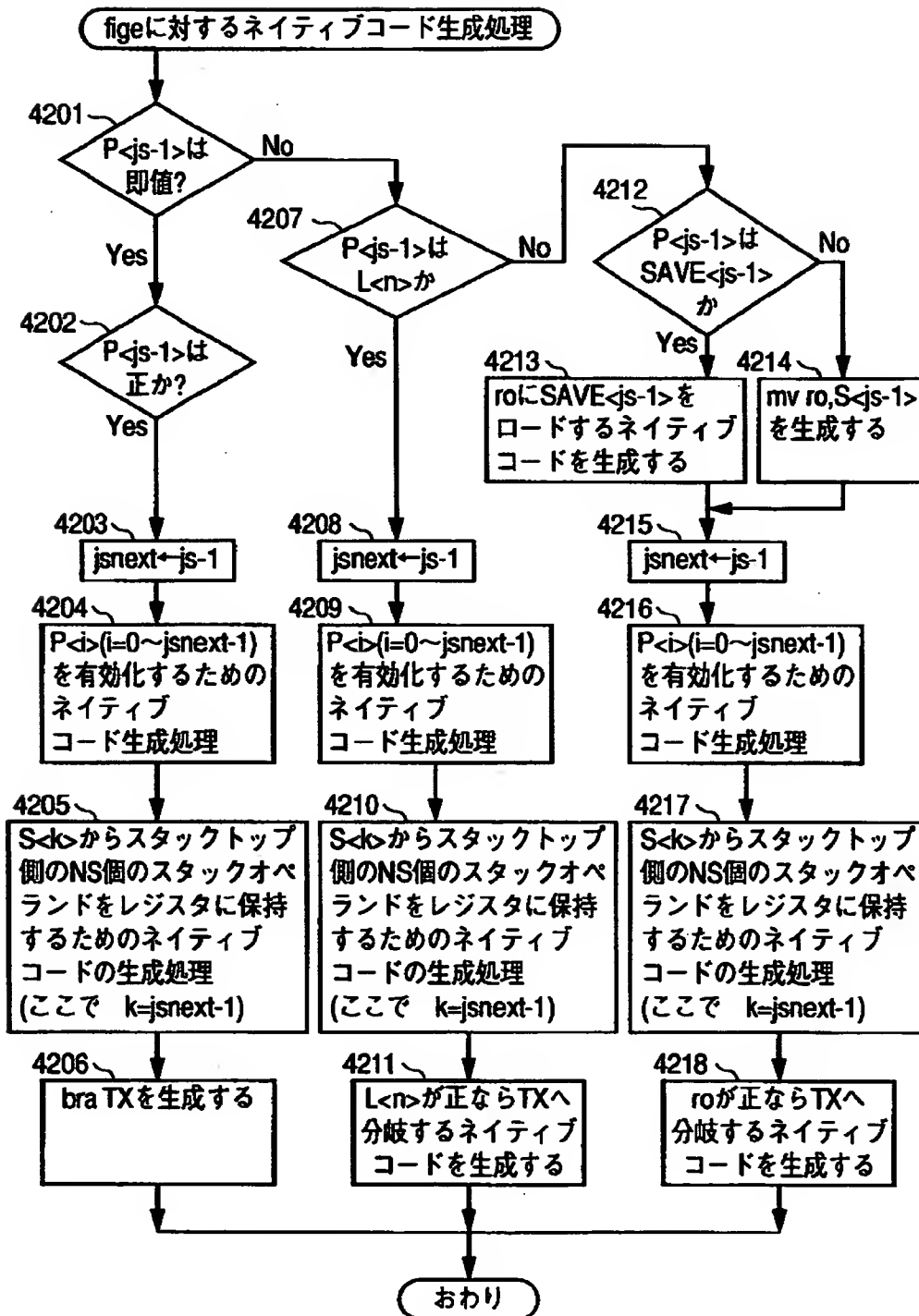
【図60】



【図 6 1】



【図 6 2】



【図 63】

行番号	命令	命令サイズ	オペランド	P<0>	P<1>	P<2>	P<3>	P<4>	P<5>	実行/コメント
(1)										addi sp, 0-(inLocal+nArg+nStack)*4 push r8 push r9 push r10 push r11 push r12 push r13 push lr ld L<0>, 0((inLocal+nStack+nSave-1)*4,sp) ld L<1>, 0((inLocal+nStack+nSave-2)*4,sp)
(2)	0 iload 0	1	1 0	L<0>	-	-	-	-	-	
(3)	1 iload 1	1	2 1	L<0>	L<1>	-	-	-	-	
(4)	2 iadd	1	3 2			-	-	-	-	
(5)	3 istore 2	1	4 1	S<0>	-	-	-	-	-	mv S<0>, L<0> add S<0>, L<1> st S<0>, 0L<2>
(6)	4 iconst 1	1	5 0	1	-	-	-	-	-	
(7)	5 iload 0	1	6 1	1	L<0>	-	-	-	-	
(8)	6 ifge 21	3	9 2			-	-	-	-	ldi S<0>, 11 bgez L<0>, r17
(9)	9 iconst 2	1	10 1	S<0>	-	-	-	-	-	
(10)	10 iload 0	1	11 2	S<0>	2	-	-	-	-	
(11)	11 iload 1	1	12 3	S<0>	2	L<0>	-	-	-	
(12)	12 iconst 3	1	13 4	S<0>	2	L<0>	L<1>	-	-	
(13)	13 iload 2	1	14 5	S<0>	2	L<0>	L<1>	3	-	
(14)	14 iadd	1	15 6			2	L<0>	L<1>	3 L<2>	st S<0>, 0SAVE<0> ld S<4>, 0L<2> add3 S<4>, S<1>, #3 mv S<3>, L<1> div S<3>, S<4> mv S<2>, L<0> add S<2>, S<3> all3 S<1>, S<2>, #1
(15)	15 ldiv	1	16 5	SAVE<0>	2	L<0>	L<1>	S<4>	-	
(16)	16 iadd	1	17 4	SAVE<0>	2	L<0>	S<3>	-	-	
(17)	17 imul	1	18 3	SAVE<0>	2	S<2>	-	-	-	
(18)	18 goto 28	3	21 2	SAVE<0>	S<1>	-	-	-	-	ld S<0>, 0SAVE<0> bra r24
(18')				S<0>	S<1>	-	-	-	-	

【図 6 4】

[illegible]

【書類名】 要約書

【要約】

【課題】 非ネイティブコードを少ないハードウェア量で高速に実行することが可能なデータ処理装置を提供する。

【解決手段】 データ処理装置は、非ネイティブコードをプロセッサのネイティブコードに変換するハードウェアトランスレータ120と、ソフトウェアトランスレータと、プロセッサに対する非ネイティブコードを逐次解釈し、プロセッサのネイティブコードを用いて実行するソフトウェアインタープリタとを含み、これらを所定の基準にしたがって選択して動作させる回路121、122などを含む。

【選択図】 図5

出 願 人 履 歴 情 報

識別番号 [000006013]

1. 変更年月日 1990年 8月24日
[変更理由] 新規登録
住 所 東京都千代田区丸の内2丁目2番3号
氏 名 三菱電機株式会社